LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TR-543

ASPECT: A FORMAL SPECIFICATION LANGUAGE FOR DETECTING BUGS

Daniel Jackson

June 1992

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 1992	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE Aspect: A Formal Specification Language for Detecting Bugs	5. FUNDING NUMBERS
6. AUTHOR(S) Jackson, D.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Massachusetts Institute of Technology Laboratory for Computer Science 545 Technology Square Cambridge, MA 02139	8. PERFORMING ORGANIZATION REPORT NUMBER MIT/LCS/TR-543
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) DARPA 1400 Wilson Blvd. Arlington, VA 22217	10. SPONSORING/MONITORING AGENCY REPORT NUMBER N00014-89-J-1988

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Aspect is a static analysis technique based on formal specifications. By trading expressive power for tractability, Aspect can offer efficient detection of a class of bugs that is not detectable by other static means. Since the specifications are partial, not all bugs can be caught. But there are never any spurious reports: an error message always indicates an error in the code or a flaw in the specification.

Aspect can handle most of the features of modern imperative programming languages: side-effects and aliasing, exceptions, polymorphism and dynamic allocation. It takes advantage of strong typing and is designed for programs that are organized around procedures and abstract types.

The checking mechanism is based on an enriched form of dependency analysis. Objects are divided into projections called 'aspects'; the dependencies of different aspects are then tracked individually. The analysis is comparable in complexity to the kinds of analysis already performed by optimizing compilers.

A prototype checker has been implemented for the CLU programming language. It runs almost as fast as the compiler, and has found a variety of bugs in real programs.

14. SUBJECT TERMS	15. NUMBER OF PAGES 162
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Aspect

A FORMAL SPECIFICATION LANGUAGE FOR DETECTING BUGS

Daniel Jackson
June 16th, 1992

This report is a copy of the author's thesis, submitted to the Department of Electrical Engineering and Computer Science in September 1992 in partial fulfillment of the requirements for the degree of Doctor of Philosophy, at the Massachusetts Institute of Technology. The thesis was supervised by Professor John V. Guttag. The research was supported by grants from DARPA (N00014-89-J-1988) and NSF (8910848-CCR). The author's current address is: School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213-3890.

Abstract

Aspect is a static analysis technique based on formal specifications. By trading expressive power for tractability, Aspect can offer efficient detection of a class of bugs that is not detectable by other static means. Since the specifications are partial, not all bugs can be caught. But there are never any spurious reports: an error message always indicates an error in the code or a flaw in the specification.

Aspect can handle most of the features of modern imperative programming languages: side-effects and aliasing, exceptions, polymorphism and dynamic allocation. It takes advantage of strong typing and is designed for programs that are organized around procedures and abstract types.

The checking mechanism is based on an enriched form of dependency analysis. Objects are divided into projections called 'aspects'; the dependencies of different aspects are then tracked individually. The analysis is comparable in complexity to the kinds of analysis already performed by optimizing compilers.

A prototype checker has been implemented for the CLU programming language. It runs almost as fast as the compiler, and has found a variety of bugs in real programs.

Acknowledgments

But for John Guttag, my thesis would be twice as long and half as good. After six years of his supervision, I think I have finally grasped what research is all about: thinking big thoughts but taking small and measured steps. He not only tolerated my iconoclastic ideas, but helped me sharpen and clarify them even when they were different from his own. Many times, after I had dug myself into a deep hole (usually because I had ignored his advice), he would announce "Let's take a step back" and half an hour later I would be back at the surface. His abilities extend beyond supervising research: if finding an academic job in a recession is like standing for President, I could not have found a more savvy and supportive campaign manager.

My readers were Butler Lampson, David Gifford and Pamela Zave. Butler complained right from the start that Aspect was too complicated (and he was right). Being subjected to his questions is like being a criminal on trial. I am glad now, though, that I was unable to fool him. and I think the thesis was immeasurably improved by his skepticism.

David Gifford gave me advice on doing experiments. His graduate programming language course, 6.821, was a highlight of my time at MIT; it gave me a foundation in type theory and semantics without which I would not have been able to write this thesis.

Pamela Zave helped me think about the broader context of my work, and her own work on specification has influenced me greatly. The summer I spent working with her at Bell in 1988 left quite an impression: I had no idea what it meant to tackle a real example until I saw her take on the specification of 5ESS almost single-handedly. I hope her writing skills also rubbed off on me. If not, at least her meticulous editing has removed many of my blunders.

My father, Michael Jackson, has been my technical cheer-leader. He has pointed me in fruitful directions by extolling the virtues of my better ideas—far beyond their merit—without ever telling me how bad the rest really are.

More than once I have faxed him a draft of a paper late at night and had his comments within hours in time for submission the next day. But most of all, I want to thank him for starting me on the long road that led to this thesis. Project Crew might have ended my computing career if he had not shown me that even COBOL is fun when treated methodically. I must say, however, that technologically he is outclassed by my mother. Her correspondence has been even more frequent since she discovered "femail" and has often been the thing I looked forward to most on arriving at work.

Keith Randall implemented the second version of the Aspect Checker and helped me invent reference aspects. Dorothy Curtis spared me some agonies trying to understand the CLU compiler by always being on hand to pore over code with me. The Two Marks were a sounding board for my ideas and, as our system managers, saved me from learning UNIX by always knowing the right incantations. Mark Reinhold managed to give me useful comments on drafts of my work in its incomprehensible phase. Mark Vandevoorde explained the subtleties of CLU to me, and his own work on object containment clarified many of the problems I faced dealing with aliasing and abstraction functions.

My work benefited from discussions with many people including: Steve Garland, Jim Horning (who dared me to start my thesis with the word "but"), Albert Meyer, Tobias Nipkow, Sharon Perl, Mark Sheldon, Raymie Stata, Yang Meng Tan, Bill Weihl, Jeannette Wing and Kathy Yelick. Many thanks also to Neena Lyall who gave me her shrikhand recipe and to Gerry Brown who supplied a constant stream of stationery and abuse.

The ITT Corporation funded my first two years of graduate school. My favorite lawyer, Claudia Marbach, funded the rest (with a little help from DARPA and NSF). If it had not been for her, Aspect would have finished me before I finished it.

In our description of nature the purpose is not to disclose the real essence of the phenomena but only to track down, so far as it is possible, relations between the manifold aspects of our experience.

Niels Bohr
Atomic Theory and the Design of Nature, 1934

Contents

List of Figures

Chapter 1

Introduction

Because programmers make mistakes, most programs contain bugs—faults that lead to malfunction. A bug may show up only under rare conditions, but its effect may be devastating[1].

Eliminating bugs is notoriously hard. Some bugs are easy to detect because they violate simple rules about the way a program should be constructed. This kind of bug, called an *anomaly*, can be detected without even knowing what the program is supposed to do. It never makes sense, for example, to read a variable that has not been assigned a value. But most bugs are not anomalies. A program free of anomalies does something right; we want to know if it does the right thing. This cannot be determined without knowing how the program is expected to behave.

One way to catch bugs is by testing: running the program on some inputs and checking the correctness of the outputs. In theory any bug can be detected in a trial run. But in practice it is hard to choose the inputs that are most likely to expose faults. Interpreting the outputs may also be tricky. Worse still, most software cannot be adequately tested until it is near completion, when bugs are harder to pinpoint and far more costly to fix.

An alternative to testing is *static analysis*: checking the program by examining, rather than executing, its text. This works best when each component has a specification—a description of its intended behaviour. A component can then be checked independently, by assuming that the components it uses meet their specifications. Whether they are faulty (or even whether their code is missing) is immaterial; the effect of bugs elsewhere can be ignored.

[1]'Car with computerized steering loses control when out of gas', *Autoweek*, June 22, 1987.

The most extensive form of static analysis is ***program verification***, in which a mathematical proof is constructed to show that a program meets its specification. Although its principles are well-understood, verification is not used in practice. For most software development projects, it is simply not cost-effective: there are far cheaper ways of eliminating bugs (such as testing). The cost of verification is so high that, safety-critical systems aside, it can rarely be justified whatever its benefits.

The cost of verification has two sources. First, even writing a complete specification is beyond the budget of most projects. To be precise enough, the specification must be *formal*, that is, written in a notation whose syntax and semantics is mathematically defined. This may not be appreciably easier than writing the code itself. Second, the proof of correctness cannot be automated; the user has to provide loop invariants and lemmas.

In contrast to verification, type checking is a form of static analysis that is widely used. Type declarations are easy to write, and in strongly typed languages (such as Pascal), can be checked automatically. They have been a spectacular success; a surprisingly high proportion of careless mistakes manifest themselves as type errors. Moreover, a program free of type errors is far easier to test. But type checking is only good for minor slips and many bugs escape it.

1.1 Aspect: A Foray into Unexplored Territory

Verification can in theory find any bug, but its cost is high; type checking is cheap but its power is limited. This thesis is an attempt to find some middle ground.

Aspect is a static analysis technique for detecting bugs in code. It comprises a specification notation and a checker that generates bug reports by finding claims in the specification that the program does not satisfy. Aspect specifications are easy to write and compositional: the specification of a composite type is constructed automatically from the specifications of its parts. The checker runs almost as fast as a compiler and without any user interaction.

The tractability of Aspect is achieved by limiting the expressiveness of its notation. Because the specifications are partial, they cannot capture all the required properties of the program, so some bugs are inevitably missed. But the checker is *sound*: it guarantees the absence of spurious reports. When a bug is listed, there must be an error in the code—or, of course, a specification flaw.

Aspect can be applied incrementally, the programmer deciding which parts of the program to specify and to what level of detail the specifications are written. The checker analyzes whatever it is given; a procedure can be checked even if it calls procedures that are not coded, coded but not specified, or neither coded nor specified.

Aspect is designed for strongly-typed, imperative languages that support data abstraction. The version described in this thesis is for CLU[2][Lis79] , but it should work for other languages with these features, such as Modula and Ada. Aspect can handle arrays and pointers, side-effects and aliasing, polymorphism, exceptions and dynamic allocation, but not higher-order functions.

1.2 Aspect in Action: A Sneak Preview

The essence of Aspect is reasoning about dependencies between the 'aspects' of abstract objects. The programmer writes a specification claiming that certain aspects of result objects depend for their computation on certain aspects of the argument objects. The checker examines the code to see if these dependencies are plausible. If one is missing—that is, a result was computed with inadequate information—an error is reported.

Figure 1.1 shows a procedure that removes duplicate elements from an array. The array is modified in place. The elements are examined in turn by incrementing *i*. The index *j* points to the end of the array prefix that is guaranteed free of duplicates; the set *s* contains the elements of this segment. The segment between *j* and *i* contains only duplicates and is trimmed off when *i* reaches the high bound.

An operation *o* of an abstract type *t* is written *t$o* in CLU. The operation *arr$low*, short for *array[int]$low*, returns the low bound from which the array is indexed. The statement *a[j] := e* is short for the procedure call

array[int]$store (a, j, e)

[2]None of the thesis assumes knowledge of CLU, but for the curious, a tutorial introduction may be found in [LG86].

```
arr = array [int]
remove_dupls = proc (a: arr)
     a_low: int := arr$low (a)
     i: int := a_low
     j: int := a_low
     s: intset := intset$create ()
     while i <= arr$high (a) do
          e: int := a[i]
          if ~intset$member (s, e) then
               intset$insert (s, e)
               a[j] := e
               j := j + 1
               end
          i := i + 1
          end
     arr$trim (a, a_low, j - a_low)
     end remove_dupls
```

Figure 1.1: Removing duplicates from an array

and its effect is to store the element e at the index j of the array a. The call *arr$trim (a,lo,ct)* removes all the elements of a below the index lo and above the index $lo + ct - 1$.

1.2.1 A Class of Simple Errors

Although the procedure is not complicated, a competent programmer might still make mistakes coding it, such as: confusing i and j by reversing their increments or writing i in the call to *trim*; misplacing the *end* of the if-statement, so that it precedes the increment of j; omitting the call to *trim* entirely, etc.

A simple observation explains why each of these mistakes must cause the program to misbehave. The size of the array after removing the duplicates should be a function of more than the size before—the values of the elements themselves must be taken into account. So there must be a path through the procedure for which the final value of the size of the array depends on the initial values of the elements. All of the slips listed above will result in a procedure that fails to satisfy this property.

Consider the case of writing i instead of j in the call to *trim*. The size of the array after the call to *trim* depends on its size before, and on the values of the second and third arguments. The actual *a_low* equals the low bound, and does not depend on the value of any element. So the only hope left for a dependence of the size on the value of an element is i. Looking at the loop, however, it is clear that i ranges only over the indices of the array.

1.2.2 How Aspect Works

Aspect finds all these bugs automatically, given only a specification of *remove_dupls*. The built-in specification of arrays views an array in terms of four 'aspects': *size*, the number of elements, *ind*, their indexing, *el*, the object identities of the elements, and *low*, the low bound. Although the array aspects may be thought of as functions on arrays (*low*, for example, mapping an array to its low bound), the checker regards them only as names of components. It needs no interpretation of these names because it only tracks dependencies between aspects and is not concerned about their values. There is one exception, however: the aspect *el* is a 'pointer' aspect that can be used to name objects that are elements of an array.

Every type is viewed in terms of aspects, but primitive types have only a single aspect. This aspect is anonymous. So *a.el*, for example, denotes the single aspect of the array element *a.el*. It is different from *el(a)*, which is an aspect of the array itself.

Here is the Aspect specification of *remove_dupls*[3]:

```
%@ size(a), el(a) ← a.el
%@ ind(a) ← ind(a), a.el
%@ el(a) :– el(a)
```

These assertions relate the values of two objects, the array *a* and the archetypal element *a.el*, over two states—before and after execution of the procedure. The expressions on the left refer to the values in the post-state, those on the right, the values in the pre-state.

The first two lines are dependency assertions: the first says that *size(a)* and *el(a)* after the execution depend on *a.el* before. This expresses formally the criterion we formulated above: that you need to look at the elements themselves (in the pre-state) to predict the final size of the array or which elements it will contain (in the post-state). The second line makes a similar claim about the array's indexing. The third line is a binding assertion. It relates the element identities before and after; its use is for alias analysis and need not trouble us here. Some assertions are implied by omission. The low bound, for example, is claimed to be invariant because there is no assertion with *low(a)* on the left.

Suppose we run the checker on the faulty variant of *remove_dupls* in which *i* is substituted for *j* in the call to *trim*. The checker would display:

```
Missing: size(a) on a.el
Missing: ind(a) on a.el
Missing: el(a) on a.el
```

showing that three required dependencies are missing. As another example, suppose we wrote *arr$bottom* for *arr$low*. The checker would then display:

```
Missing: low(a) on low(a)
```

[3]%@ is a special comment symbol that distinguishes Aspect specifications from informal comments.

because the implied invariance of *low(a)* requires a dependence of *low(a)* on itself. It is not found in this case because the call to *trim* resets the low bound to the bottom element of the array and not its index.

To find the missing dependencies, the checker calculates an approximation to the entire dependency relation between the aspects of the pre- and post-states and checks that it contains the dependencies specified. The approximation is an over-estimate: it may include dependencies for paths that cannot occur in any execution. The lack of precision may let a bug slip through, but it cannot cause spurious messages, because if a bug is missing from the over-estimate, it must be missing from the exact dependency relation too.

Calculating Aspect dependencies is like calculating regular dependencies between variables. For example, the dependencies of

```
x := y
if b then z := x end
```

are: x on y, z on b (since whether z is changed depends on b), z on y, z on z, and b on b. A procedure call is usually handled by assuming that each argument after depends on every argument before. Aspect differs by separating the dependencies of an object's aspects and by using a procedure's specification when it is called. The array operation *arr$low*, for example, has a built-in specification

```
low = proc (a: arr) returns (int)
    %@ result ← low(a)
```

that Aspect uses to give the first line of *remove_dupls*

```
a_low := arr$low (a)
```

a dependency of *a_low* on *low(a)* and the remaining (invariant) aspects dependencies on themselves: *size(a)* on *size(a)*, *low(a)* on *low(a)*, and so on. A call to *remove_dupls* would be treated in the same way, using the same specification that is used to check *remove_dupls* itself.

1.3 Foundations

The Aspect dependency calculation is similar to the kind of dataflow analysis already performed by optimizing compilers [Fer87]. It relies on strong typing:

the type checker provides the type of an object, which Aspect uses to divide the object into aspects. Aspect performs an alias analysis that is most similar to Larus and Hilfinger's scheme [LH88] but was developed independently. The basic principles of this kind of alias analysis can be formalized as an abstract interpretation [HPR89]. I avoided more elaborate aliasing schemes (e.g. [CWZ90]) because they would complicate the specification notation without much gain in precision.

An early conversation with Dewayne Perry got Aspect started. He told me about his discovery that a large number of bugs in a telephone switch were interface errors that could have been detected with simple, tractable specifications [Per87]. His idea of "constructive specification" is embodied by Inscape [Per89], whose relation to Aspect is discussed in Section 9.2.

Aspect is built on the notions of declarative specifications and abstract data types developed by many researchers, including Guttag [Gut77], Parnas [Par72] and Hoare [Hoa72].

1.4 A Brief Survey of Bug-Elimination Schemes

Section 9.2 compares Aspect to its most closely related research. This brief survey sets Aspect in a broader context.

1.4.1 .\un-time Assertions

The Aspect analysis is performed entirely at compile-time. In contrast, some systems (e.g., Euclid [Lam81], ANNA [LvH85] and Eiffel [Mey88]) allow programmers to write assertions in the program text that are checked at run-time. Because assertions must be executable, they are laborious to write and there is a risk that the programmer will express an assertion with a procedure from the code itself. This reduces the redundancy value of the assertion, and raises the awful spectre of perfectly good code crashing because of an assertion. In ANNA, for example, only convention prevents an assertion from having a disruptive side-effect. Runtime assertions also incur a performance penalty, tempting the developer to switch off the checking when the system is being used and, arguably, needs it most. Lastly, ru⁻-time assertions only work on a finished program: a procedure cannot be ruıı until the procedures it calls have been coded too.

Consider writing run-time assertions for *remove_dupls*. One obvious assertion says that the array is unchanged when viewed as a set:

$$array\_to\_set\ (a\_pre) = array\_to\_set\ (a\_post)$$

but this does not help much, because if we cannot code *remove_dupls* correctly, we are unlikely to do better on *array_to_set*. More abstract assertions do not spring to mind. Since run-time assertions are confined to single executions of a procedure, they have no analogue to an Aspect dependency like

$$size(a) \leftarrow a.el$$

that makes a claim about a set of executions. It would be no good to assert, say, that every execution looks at some element of a; that would be violated by the correct code when called on an empty array.

Despite these problems, run-time assertions are indispensable[4].. It is usually less work to annotate a program than to test it exhaustively. They are especially helpful in locating bugs. When an assertion is violated, the checker can behave like a debugger, printing out the values of variables and identifying the point at which the violation occurred.

1.4.2 Verification

Verification is a useful technique for assuring the correctness of intricate algorithms; its use in circuit analysis (for chip design) looks promising too. But for software development it does not seem feasible. The problem is one of scale, but not in the usual sense; if it worked for procedures, then it might indeed scale to systems. Unfortunately, complex data types and indirection in the store make a full proof of correctness quite complicated even for a 50-line procedure.

Some proponents of verification (e.g., [Dij76]) have argued that this complexity can be overcome if the program is constructed hand-in-hand with its proof. This has been elegantly demonstrated for some small algorithms[5].

[4]And I used them in the implementation of the Aspect checker itself.

[5]Correctness by construction is not infallible; some of these demonstrations appear in a collection of published programs that were found to have bugs despite being verified [GY76].

In a sense, Aspect checking is a crude form of verification. Like the formulae of a Hoare triple, Aspect assertions are implicit—relating the post-states to the pre-states rather than giving a recipe for their construction—and quantified—defined over the set of all possible executions rather than a single execution.

There is another sense, however, in which Aspect is not verification. A conventional specification is universally quantified: it says that some property holds for every execution. An Aspect specification is existentially quantified; it says that executions exist with certain dependencies. This is why an Aspect dependency cannot be converted into a run-time assertion.

Aspect is designed this way because it is much harder to make a checker sound if the specifications are universally quantified. With a conventional specification, demonstrating the existence of a bug is equivalent to finding an execution that violates a universal property. In rare cases, it might be possible to show that all paths through the code violate the property, but most often the violation will occur only on some path. To guarantee no spurious messages, the checker must prove that this path can actually occur—*in general, an undecidable problem.*

Tractable analyses based on partial verification have two choices. They can simply forbid all programs that appear to be faulty. Type checkers do this, flagging type errors in dead code for instance. The alternative is to give up soundness and generate spurious messages. This has been the approach of anomaly detectors (Section 1.4.5) like Lint and DAVE.

I decided to make Aspect sound because I do not believe that either of these choices is acceptable. Forbidding unprovable programs is too extreme once arbitrary specifications can be written. Spurious messages weaken the utility of the checker; Lint is spurned by many programmers because so few of its messages signal real problems. Rumour has it that a recent software catastrophe[6] was due to a bug that Lint would have caught, but it was not used because the cost of poring over reams of bogus messages was deemed too high.

[6]A crash of the AT&T phone system on January 15th, 1990. See *ACM Software Eng. Notes, 15/2*, April 1990.

1.4.3 Testing

There is no substitute for testing; to make sure that a program works, you have to try it. But testing alone is not sufficient. Like run-time assertions, it only works on finished programs. One can write stubs for unimplemented procedures, but this is laborious and itself error-prone. Furthermore, it is not always easy to evaluate the results of a test run. This problem is particularly severe for testing interactive software where special tools (often tailored to the system being tested) are needed to record and replay test scripts. And lastly, there is the problem of selecting test cases. The state space of a program is generally so big that running a huge set of tests is not enough—they have to be carefully picked. This is a troublesome and costly exercise. Some ingenious techniques, such as coverage analysis [How76] and mutation testing [Acr79, Off92], have been proposed to determine whether a program has been adequately tested, but they do not work well on code that is implemented with complex abstract types.

With the right test case, any bug can be found. Aspect, though, can only catch a class of fairly simple errors. For these it is likely to be cheaper and more effective. The cost of constructing and evaluating a test case is the same whether it is designed to catch a simple or a subtle bug. For example, a test that *remove_dupls* maintains the emptiness of the array would require sample arrays no easier to construct than for any other test case. If the procedure removed duplicate keys from a table represented as an array of pairs, the samples would be more complicated, even though the internal structure of the array is immaterial. Aspect's benefit comes more in proportion to its cost; a simple check needs only a simple assertion.

Symbolic testing [How77] is a hybrid of testing and verification. The program is executed symbolically on test cases that are expressions containing free variables. Although each expression represents a set of conventional test cases, the user has to specify which path to take at a conditional branch and how many times to go round a loop. A more elaborate approach [CHT79] using an expression simplifier overcomes this problem, but its results may be unsolved recurrence relations. These techniques cannot incorporate specifications or abstract types, and seem to be suited only to numerical programs.

1.4.4 Type checking

Type checking is limited in the kinds of bug it can catch. Calling the wrong procedure, using the wrong variable name or misordering the arguments to a procedure may all lead to type errors. But since the type of an object is fixed over its lifetime, checking is insensitive to changes of state and has no hope of catching omissions or misorderings of statements.

Type checking differs fundamentally from Aspect in two respects. First, type checking addresses errors of commission rather than omission. A procedure that contains no code is always type-correct, but it will satisfy only one Aspect specification—the empty one. Second, a type is a property of a single object, so type checkers cannot detect errors in the way objects are put together. Aspect, in contrast, is all about dependencies between aspects of objects. We saw in the *remove_dupls* procedure, for example, how the checker can catch an error of substituting i for j in the call to to *trim*. One carried a dependency on the elements of the array; the other did not. Aspect can distinguish these two integers in the way they relate to the array elements.

Type declarations often make fewer distinctions than Aspect specifications. For example, the array operations *size* and *low* have the same type signature, but the Aspect specifications differ: for an array a, the results depend on *size(a)* and *low(a)* respectively. Unlike Aspect, the type checker cannot distinguish the operations *bottom* and *low*, because the bottom element is an integer like the low index. Type abstraction and polymorphism do make type checking a little more discriminating though; if the array were polymorphic, *bottom* and *low* would indeed have distinguishable signatures.

The relationship between Aspect and more sophisticated type systems (such as Typestate [Str83], FX [GJSO92] and ML refinement types [FP92]) is discussed in Section 9.2.

1.4.5 Programming Language Design

Type checking is only one example of a general trend. A primary goal in programming language design has been to limit the kinds of bug that go undetected. This has been achieved in four ways: by removing features of the language that are notoriously hard to use correctly, by providing extra features that simplify code, by catching errors during the execution and by detecting flaws at compile-time.

CLU, for example, eliminates global variables, stack allocation of abstract objects and manual deallocation of objects. These are responsible for many initialization errors[7]. It provides exception handling—a major advance over checking return status; iterators, which allow operations to be performed on the elements of indexed structures without any explicit indexing; and dynamic arrays, which are designed so they cannot contain "holes". Variables can be declared at the point of use, which further reduces the incidence of initialization errors. The run-time system detects array bound errors, arithmetic overflow and unhandled exceptions. The type system is strong, so all type errors are caught by the compiler, and has no loopholes[8].

CLU's most significant design feature is its support of data abstraction. Localizing and encapsulating the representation of an abstract type simplifies the programs that use the type. The CLU compiler restricts access to a type's representation to a select set of procedures[9] and distinguishes different abstract types with the same representation.

Older programming languages, and languages like C that are intended for low-level applications, cannot detect or prevent most of the bugs that are ruled out by CLU. Free-standing tools have been built to find anomalies (flaws that are likely to be bugs) in these languages. DAVE [FO76], designed for Fortran, used regular expressions to express the orders in which elementary operations could occur; its ideas have been applied more recently to C [WO85]. Bergeretti and Carre's technique [BC85] is similar, finding discrepancies in a dataflow matrix relating all the instances of variables and expressions in a procedure.

Cesar [OO89] is an extension of DAVE that allows the user to define path expressions for the permissible sequencing of operations. This kind of analysis applies to modern languages too, but it is not clear how frequent sequencing errors of this kind are for user-defined operations. Primitive objects may have a stylized pattern of usage—a read-only file must be opened, read and then closed—but the operations of a set, table, queue or stack are not so constrained. The more serious problems of Cesar are its spurious messages and the impractical global analysis it requires. These issues are revisited, along with a discussion of a related technique [How90], in Section 9.2.

[7]Liskov explains why in [Lis92].

[8]For example, variants do not cause trouble as in Pascal, because of the way objects are designed.

[9]Most of the time. 'Rep exposure' can beat the type checker [LG86].

1.5 The Organization of the Thesis

Chapter 2 explains the basic idea of dependencies between aspects. A procedure specification plays two roles: in checking its own code and in checking the code of its callers. To check *remove_dupls*, for example, the checker uses not only the specification of the *remove_dupls* procedure itself, but also the specifications of the built-in operations + and *array[int]$low*.

Chapter 3 introduces pointers and container objects (like arrays and sets). Pointers lead to aliasing, which wreaks havoc with the dependency analysis. The problem is solved with a special kind of aspect to represent pointers, and a second kind of assertion called a 'binding' that says how a procedure may change the shape of the store.

Chapter 4 refines the definitions of the assertions introduced in Chapters 2 and 3. It exposes some implications that have been hidden until this point. By being careful about the interpretation of object names in assertions, we see how interactions between assertions can be avoided so they retain their declarative meaning. Refining the definitions of the assertions also allows the checker to catch more bugs than the simplified presentation of Chapters 2 and 3 suggests.

Chapter 5 explains abstraction functions. The specifier of an abstract type invents a set of aspects and uses them to specify each abstract operation. This allows clients to be analyzed in abstract terms that are independent of how the type is implemented. To check the code of the abstract operations themselves, though, the checker needs to know how the abstract aspects are related to the aspects of the representation. The chapter explains how this information is provided and what the checker does with it.

Chapter 6 formalizes the ideas of the previous chapters. The meaning of the assertions is defined in terms of transitions over abstract states. The program constructs of CLU are defined in the same way, so that whether a procedure's code meets its specification is reduced to a mathematical conjecture.

Chapter 7 describes the mechanism of the checker. Even within the Aspect model, approximations are made; the checker executes the code over states that each represent a set of Aspect states.

Chapter 8 starts by relaxing two constraints that are assumed up to this point: that the aspects of a type be independent and that specifications of called procedures be provided. It then discusses some limitations of Aspect and speculates on how they may be overcome.

Chapter 9 concludes the thesis. It describes my experience using Aspect, compares Aspect to related work and summarizes its contributions.

The essence of the thesis is conveyed in Chapters 2, 3 and 4. Chapter 6 contains nothing new: it merely formalizes the ideas of previous chapters and fills in some details that had been left to intuition. I have also tried to avoid backward references in the concluding chapter, so there is something to be gained from reading the introduction and conclusion alone.

Chapter 2

Aspect Dependencies

This chapter explains the basic principle of Aspect's bug detection scheme. The programmer specifies a division of the objects of each type into abstract components called "aspects", and then gives, for each procedure, a set of dependencies that are required to hold between the aspects of the pre- and post-states. The checker finds dependencies that are required but missing from the code. A missing dependency means that a result is computed without adequate information and that there must thus be a bug (or a specification flaw).

A series of tiny examples based on an editor buffer illustrates these ideas. We shall see how each procedure specification plays two roles but has only a single meaning. Then we look at a more complicated example in some detail to understand how one writes an Aspect specification and what sort of bugs can be caught. Finally, we see how aspect dependencies take account of control flow too.

2.1 Editor Buffers

An editor is a program for writing and modifying documents. The text of the document being edited resides in an object called the *buffer*, part of which is displayed on the screen. The user makes changes to the document by altering the buffer and then saving it in a file.

The buffer has a *cursor* indicating the point at which a character typed at the keyboard will be inserted. It also has a *mark*, a kind of secondary cursor usually not visible. The purpose of the mark is to delineate with the cursor a portion of the text called the *region*. There are special commands that operate on regions, such as "delete-region", which deletes from the buffer the characters between the mark and the cursor. The mark is set by placing the cursor at the desired position and issuing the command "set-mark-at-cursor"; the cursor is then moved elsewhere to define the region.

Finally, the buffer has a *clipboard* that contains a fragment of text cut from the document to be reinserted later. The "cut-region" command deletes the text in the region and places it in the clipboard (thereby eliminating whatever was there before). By moving the cursor and issuing the command "paste", one can reinsert the cut text in a different place. The clipboard is considered to be part of the buffer and not a separate object.

2.2 First Example: A Detectable Bug

Suppose we have an abstract type *buf* that models an editor buffer and provides operations like ***buf$get_mark*** for obtaining the current position of the mark, ***buf$set_cursor*** for setting the position of the cursor, etc. Here is a procedure that uses these operations:

```
exchange = proc (b: buf)
    c: int := buf$get_cursor (b)
    buf$set_mark (b, c)
    m: int := buf$get_mark (b)
    buf$set_cursor (b, m)
    end exchange
```

It is intended to exchange the positions of the mark and cursor, but it fails for a simple reason. By resetting the mark before saving its old value, the procedure sets the cursor to the new mark position instead of the old one, thus leaving the cursor unchanged.

We can formulate this reasoning in Aspect as follows. The final position of the cursor should depend on the initial position of the mark, but there is no such dependency in the code. The Aspect specification of the procedure is:

```
exchange = proc (b: buf)
    %@ mark(b) ← cursor(b)
    %@ cursor(b) ← mark(b)
```

It contains two *dependency assertions*. The expressions on the left refer to the post-state and those on the right to the pre-state. The first assertion says that the mark of the buffer in the post-state should depend at least on the cursor in the pre-state; the second assertion says that the post-state cursor depends on the pre-state mark. In response to the procedure annotated with this specification, the checker would display the message:

Missing: cursor(b) on mark(b)

In contrast, the dependency of the first assertion is found and so no message is produced for it.

The checker cannot perform this analysis with the code and its annotation alone—it must know how the called procedures affect the cursor and mark. This information is derived from the Aspect specification of the buffer type.

2.3 The Specification of the Buffer Type

Part of the specification of the buffer type is shown in Figure 2.1. Immediately following the header, the four aspects of a buffer are listed: *text*, *clip*, *cursor* and *mark*. Every buffer is subsequently viewed in terms of these four aspects. The text aspect of a buffer *b* is written *text(b)*, and denotes the text contained in *b*; *clip(b)* denotes the contents of its clipboard, and so on.

Each operation of the type is given a specification just like the specification of *exchange*: a claim that certain minimal dependencies hold between the aspects of the pre- and post-state.

The *set_mark* specification, for instance, says that the value of the mark of *b* in the post-state depends on the value of the argument *i* in the pre-state. Compare this to the specification of *move_mark*, which moves the mark by *i* rather than setting the mark to *i*, and thus has a dependency of mark on its old value.

The omission of assertions for the other aspects implies that they are invariant, so the specification of *set_mark* implies that setting the mark does not affect the text, clipboard or cursor. Similarly, since no aspect of *b* appears on the left-hand side of an assertion of *get_mark*, we can infer that the entire buffer *b* is unchanged.

The last operation of Figure 2.1, *reset_cursor*, resets the cursor to the top of the buffer. Since *cursor(b)* is set to a constant, no dependences are given

```
buf = cluster is set_cursor, set_mark, get_mark, get_cursor, ...

    %@ aspects text,      % textual contents of buffer
    %@         clip,      % contents of clipboard
    %@         cursor,    % displacement of cursor from top of buffer
    %@         mark       % displacement of mark from top of buffer

    set_cursor = proc (b: buf, i: int)
        %@ cursor(b) ← i

    set_mark = proc (b: buf, i: int)
        %@ mark(b) ← i

    get_cursor = proc (b: buf) returns (int)
        %@ result ← cursor(b)

    get_mark = proc (b: buf) returns (int)
        %@ result ← mark(b)

    move_mark = proc (b: buf, i: int)
        %@ mark(b) ← i, mark(b)

    reset_cursor = proc (b: buf)
        %@ cursor(b) ← ∅
```

Figure 2.1: Part of the buffer specification

and the special symbol $\emptyset$, denoting the empty list of aspect expressions, is used.

Note that the specification nowhere defines a "meaning" for any of the aspects. An aspect is just a name for a component of an object. The interpretation of the aspects recorded informally in the comments alongside their declaration is not part of the Aspect specification. The specification does not even attribute types to the aspects, so although one can think of *mark*, for example, as a function that can be applied to buffers, the result of applying the function is not typed. To type-check the specifications, the checker gives *mark* the signature

$$mark : buf \longrightarrow AspectRange$$

which allows it to reject an expression of the form $mark(x)$ if x is not a buffer. All aspect expressions have the type *AspectRange*, so a dependency assertion $\alpha \longleftarrow \beta$ always type-checks when α and β type-check. But since no other type matches *AspectRange*, nesting of aspect expressions is not allowed. Even if we had an abstract type cursor with an aspect *atStart*, for example, we could not form aspect expressions like $atStart(cursor(b))$.

The expression i denotes the value aspect of the integer i. When a type is not divided into aspects, its objects are considered to have a single "value" aspect that has no name. Like the buffer aspects, the value aspect of integers can be viewed as a function with the signature

$$value : int \longrightarrow AspectRange$$

Thus, when the expression i appears in an assertion, its type is *AspectRange*. It would *not* be a type error to write, for example,

$$text(b) \longleftarrow i$$

and later we shall come across such assertions.

No interpretation is given for the aspects because none is needed. The specifications make claims only about dependencies between aspects, and these can be evaluated without recourse to their values.

2.4 The Lack of Interpretation of Aspects

The lack of a given interpretation for the aspects is the key to Aspect's economy. A conventional specification would give a *model* for the abstract-type, representing a buffer as a sequence of characters, for example. The specifications of the operations would then be in terms of this model; the more complicated the model, the more complicated the specifications.

There are principally two costs of the modelling: the writing of the specifications and their analysis. Just writing a buffer specification is a difficult exercise, challenging enough to be the subject of research papers in specification[1]. Analysis is expensive because it cannot be automated. To prove any interesting property of a sequence of characters will require the skills of a mathematician: inventing lemmas and finding proof strategies. Aspect specifications, on the other hand, are relatively easy to write and code checking is automatic.

2.5 The Two Roles of Specifications

Specifications play two roles. The specification of *exchange* is a yardstick against which its implementation may be judged. In contrast, the specification of a buffer operation like *get_cursor* is used to summarize its behaviour when checking procedures that call it, like *exchange*. All Aspect procedure specifications can play both roles.

There are many reasons why a specification is used as a summary of a procedure's dependencies in preference to its code. First, the code may simply not be present, either because it has not yet been written, or because the procedure is built-in. Second, the code might be wrong; we want to find bugs in a procedure independently of the bugs in the procedures it calls. Third, an operation of an abstract type will be implemented with some other type whose aspects are different: the aspect *mark*, for instance, might not represented directly in the representation of a buffer. This issue is discussed in Chapter 5.

2.6 Specifications Express Extra Information

These reasons for preferring the specification of a called procedure to its code would apply to any specification language. Another reason arises for Aspect specifications in particular: a specification of a procedure may include information that the checker cannot infer from its code.

[1] [Suf82], for example.

Suppose *exchange* were implemented with a call to a new procedure *set_mark_at_cursor*:

```
exchange = proc (b: buf)
    %@ mark(b) ← cursor(b)
    %@ cursor(b) ← mark(b)
    set_mark_at_cursor (b)
    m: int := buf$get_mark (b)
    buf$set_cursor (b, m)
    end exchange
```

Here is the code for the called procedure:

```
set_mark_at_cursor = proc (b: buf)
    c: int := buf$get_cursor (b)
    m: int := buf$get_mark (b)
    buf$move_mark (b, c − m)
    end set_mark_at_cursor
```

The checker could not find a bug in *exchange* given only the code of the two procedures, even though *exchange* has the same bug as before. The reason is that the dependency previously missing (of the new *cursor* on the old *mark*) is actually present, because of the way *set_mark_at_cursor* is implemented. The expression *c - m* depends on the values of *cursor* and *mark* before the call to *set_mark_at_cursor*, and so the *mark* that results from the call will depend on the initial *mark*. The missing link that gave away the bug before has now reappeared.

Suppose, however, that we provide a specification for the called procedure:

```
set_mark_at_cursor = proc (b: buf)
    %@ mark(b) ← cursor(b)
```

The checker now finds the bug in *exchange* and prints out the message

```
Missing: cursor(b) on mark(b)
```

The specification of *set_mark_at_cursor* tells the checker that the dependency of *mark(b)* on *mark(b)* is accidental; only the dependency on *cursor(b)* is required. As a result, one of the specified dependencies of *exchange* is now found to be missing, since it is no longer masked by the extra dependency of *set_mark_at_cursor*.

2.7 Two Interpretations of Dependency Assertions

The checker interprets the specification of a called procedure differently from the specification of a procedure being checked. In our example, the specification of *exchange* gives minimal dependencies, whereas the specification of *set_mark_at_cursor* gives maximal dependencies.

These are two sides of the same coin. A specification of a procedure says that an implementation is valid if it has at least certain dependencies. In calling the procedure, the checker can assume it has no more (since a valid implementation is required only to have the minimum), but in checking the procedure, it can demand that there be no fewer.

A valid implementation may have more dependencies than its specification requires. The specification of *set_mark_at_cursor*, for instance, only requires *mark(b)* to depend on *cursor(b)*, and yet the implementation—which is correct—has an extra dependence of *mark(b)* on *mark(b)*. In analyzing *exchange*, however, the checker assumes such dependencies are missing. The code of *exchange* must work for any valid implementation of *set_mark_at_cursor*, and it is mere accident that *this* implementation has extra dependencies.

The reader may be troubled by this argument. A full specification of *move_mark (b, i)* would give the final position of the mark as $m + i$, where m is its initial position. If the increment i is $c - m$, the resulting mark position is

$$m + c - m = c$$

which is independent of the initial position. Why then do we say that *setMarkAtCursor* has a dependency of *mark(b)* on *mark(b)*?

Dependencies in Aspect are a syntactic notion. When we talk of a dependency of α on β, we mean that there is some path through the code in which the value of β is read and used in the computation of α. This syntactic dependency does not imply a semantic dependency—that a change in the initial value of β can affect the final value of α. Nevertheless, the converse is true: there can be no semantic dependency without a syntactic dependency. This is what makes the Aspect checker sound.

In terms of semantic dependencies, a procedure specification gives the exact dependencies that are required. The code may have no more and no fewer. The checker, however, can only approximate the semantic dependencies. Its approximation is an upper bound, so it can only conclude an error when there are fewer apparent dependencies than required. In attempting to find missing dependencies, it infers from the specifications of called procedures that some dependencies are not present, and from the specification of the procedure being checked that some must be present.

We shall see later that part of the difficulty in approximating semantic dependencies accurately is that there may be several paths through the code. Some of these may not be possible, but the checker cannot rule them out, since it does not have enough information to evaluate conditionals. The above example illustrates, though, that the problem arises even in straight-line code. Exact semantic dependencies cannot be calculated without full specifications of called procedures.

2.8 Second Example: More Detail

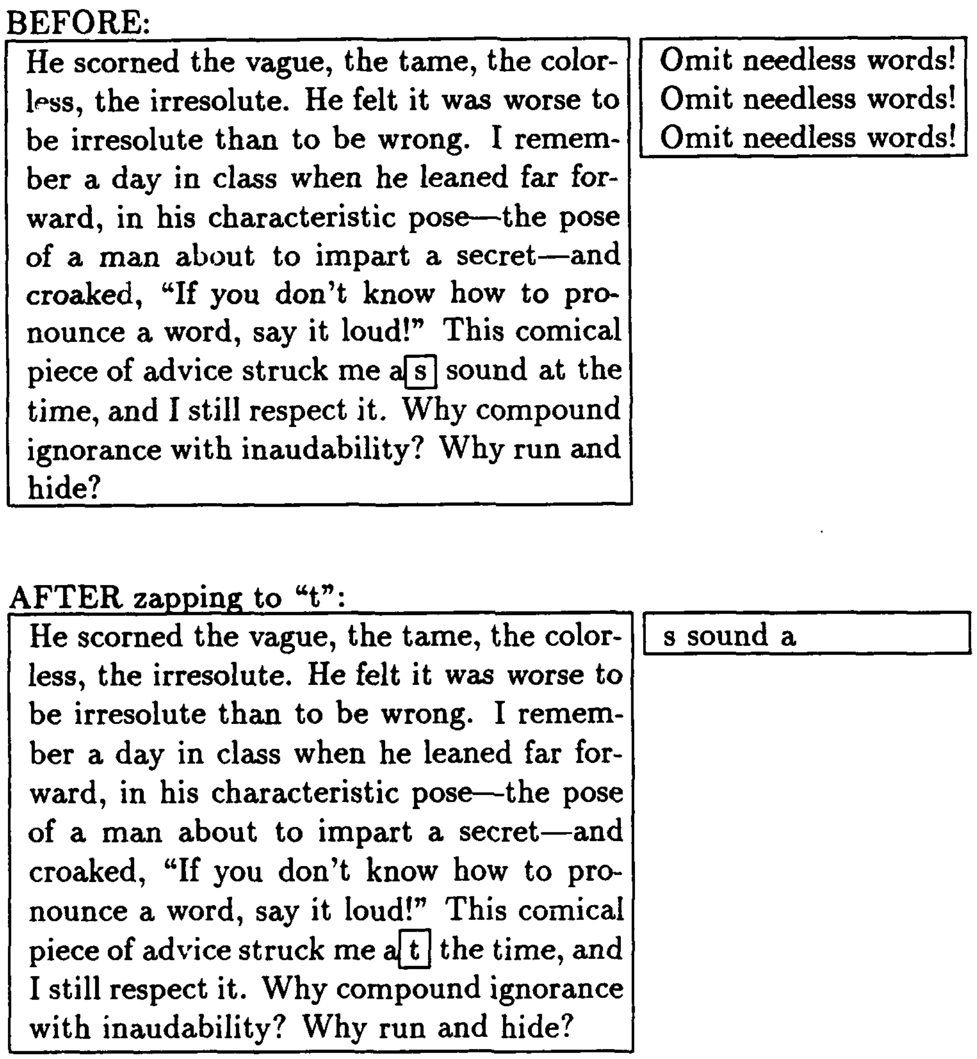
The bug in the *exchange* procedure was detectable because of a missing dependency of the cursor on the mark. These happen to be represented as integers in the procedure, but this has no relevance to the Aspect checking. To make this point clear, we now look at an example that uses the other buffer aspects, *clip* and *text*.

The Aspect analysis will be seen to be exactly the same as that performed for *cursor* and *mark*, no more complex in its semantics nor less efficient in the checking computation. This should illustrate more convincingly the economy gained by treating aspects as components without values. In a conventional treatment of correctness, going from assertions about integers to assertions about text fragments would entail a leap in complexity; in Aspect, it does not.

Consider a procedure *zap* with the signature

zap = proc (b: buf, c: char)

that deletes the portion of the text in the buffer *b* between the cursor position and the next instance of the character *c*, and places this "zapped" text in the clipboard. Figure 2.2 shows a zapping scenario. The text is the large box on the left and the clipboard is the smaller box on the right; the cursor precedes the character with the tiny box around it.



BEFORE:

He scorned the vague, the tame, the colorless, the irresolute. He felt it was worse to be irresolute than to be wrong. I remember a day in class when he leaned far forward, in his characteristic pose—the pose of a man about to impart a secret—and croaked, "If you don't know how to pronounce a word, say it loud!" This comical piece of advice struck me a$\boxed{\text{s}}$ sound at the time, and I still respect it. Why compound ignorance with inaudability? Why run and hide?

Omit needless words!
Omit needless words!
Omit needless words!

AFTER zapping to "t":

He scorned the vague, the tame, the colorless, the irresolute. He felt it was worse to be irresolute than to be wrong. I remember a day in class when he leaned far forward, in his characteristic pose—the pose of a man about to impart a secret—and croaked, "If you don't know how to pronounce a word, say it loud!" This comical piece of advice struck me a$\boxed{\text{t}}$ the time, and I still respect it. Why compound ignorance with inaudability? Why run and hide?

s sound a

Figure 2.2: A zapping scenario

How should we specify *zap*? One way to start is to list the aspects of the argument objects. Characters have only a single aspect, which therefore has no name, and so c is the sole aspect of the variable c. The buffer aspects are

cursor(b), mark(b), text(b), clip(b)

Zapping does not move the cursor. It ends up on a different character because text ahead of it is deleted, not because its position with respect to the top of the buffer is moved. The character c is certainly not modified[2]. There are thus only two aspects that change: *text(b)* and *clip(b)*. The clipboard's new contents are the zapped text, which is defined by the character c, the position of the cursor and the text of the buffer, and so we write:

clip(b) $\leftarrow$ *text(b), cursor(b), c*

Note that the aspect *clip(b)* in the post-state does not depend on its value in the pre-state, because the zapped text replaces rather than extends the old contents of the clipboard. The text of the buffer, on the other hand, is altered by the deletion, but the rest of the document is retained, so we have:

text(b) $\leftarrow$ *text(b), cursor(b), c*

Here now is an attempt to implement zapping in a procedure that makes use of existing buffer operations, annotated with our specification:

```
zap = proc (b: buf, c: char)
    %@ clip(b), text(b) ← text(b), cursor(b), c
    k: int := buf$search (b, c)
    buf$set_mark (b, k)
    buf$cut_region (b)
    end zap
```

The assertions of *clip(b)* and *text(b)* have been joined since they share the same dependencies. The shortened form is identical in meaning to the two individual assertions.

The code works by setting the mark to the zap character, so that the region contains the text to be zapped. A call to *cut_region* is then enough to delete the zapped text from the document and copy it to the clipboard. In fact, however, the code does not satisfy its specification, and the checker would find a bug. We now take a look in more detail at how the bug is found.

[2]And in fact, since characters are immutable in CLU, it would make no sense to claim that c changes.

```
buf = cluster is ..., search, cut_region, del_region, ...
  %@ aspects text, clip, cursor, mark

  search = proc (b: buf, c: char) returns (int)
    %@ result ← cursor(b), text(b), c

  cut_region = proc (b: buf)
    %@ mark(b), cursor(b) ← mark(b), cursor(b)
    %@ clip(b), text(b) ← text(b), mark(b), cursor(b)

  del_region = proc (b: buf)
    %@ mark(b), cursor(b) ← mark(b), cursor(b)
    %@ text(b) ← text(b), mark(b), cursor(b)
```

Figure 2.3: More of the buffer specification

2.9 Calculating Dependencies

We shall need the specifications of the other buffer operations (Figure 2.3). Note that the set of aspects

text(b), mark(b), cursor(b)

appears in both *cut_region* and *del_region*, since it represents the region. But *del_region* deletes the region from the text and leaves the clipboard unchanged, so unlike *cut_region* it gives no assertion for *clip(b)*.

The dependencies of the code are constructed using these specifications as follows. First, the checker divides the argument objects *b* and *c* into aspects according to the specifications of their types (Figure 2.4). The five aspects of the argument objects (four for *b* and one for *c*) are then assigned tags, so that we can trace the flow of these aspects through the code. Integers are convenient, but any set of distinguishable labels would do.

The checker performs a kind of symbolic execution of the procedure. To find the result of the first statement

```
k: int := buf$search (b, c)
```

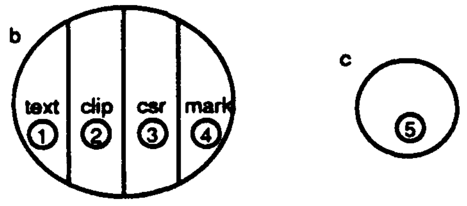


Figure 2.4: Dividing arguments into aspects

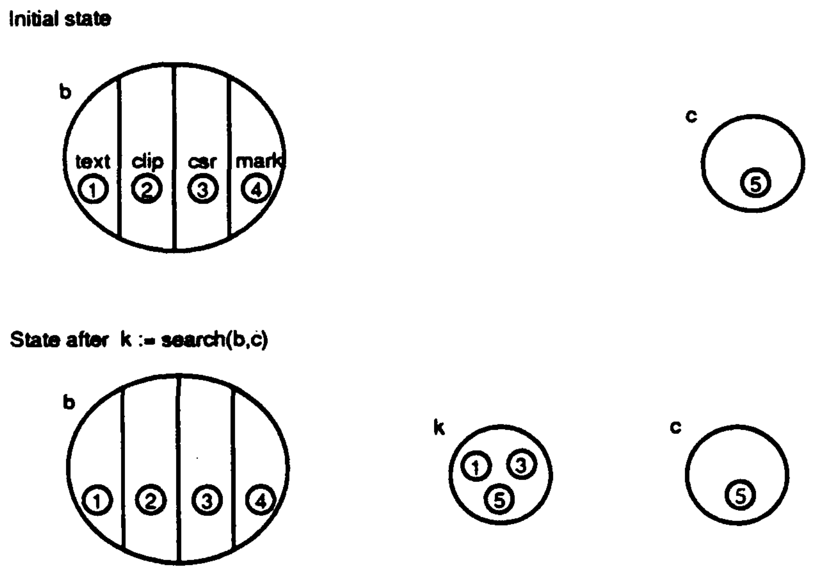


Figure 2.5: The first state transition of the abstract execution

it looks up the specification of the *search* operation and constructs the new state shown in the second row of Figure 2.5. The integer k has been allocated and tagged to show its dependence on the aspects of the initial state. The tags are obtained simply by "executing" *search*'s dependency assertion

$$result \leftarrow cursor(b),\ text(b),\ c$$

which can be read operationally as "give the (sole aspect of the) result the tags of *cursor(b)*, *text(b)* and *c*". The aspects of b keep their old tags, since there are no assertions for b (it being invariant).

A dependency is not really a property of an individual state, but a relationship with an earlier state: here, the initial state. And yet the tagging of a state allows us to think of the state's dependences on the initial state as a kind of abstract value. This is known (in abstract interpretation jargon) as "instrumenting" the state. It is convenient for reasons that will not become clear until later[3].

The remaining two statements are executed similarly (Figure 2.6). Note how *set_mark* leaves all aspects but the *mark* invariant and how *cut_region* introduces the dependence of the *text* on *c*. The final state of the execution represents the dependences of the aspects of the post-state on the aspects of the pre-state. All that remains is to assess it against the specification.

To do this, we construct a post-state from the specification of *zap*, in the same way that we "executed" the specifications of the buffer operations. The resulting state is shown with the final state of the code in Figure 2.7. To find the missing dependencies, the checker takes each aspect of the required post-state in turn and subtracts from its set of tags the tags of that aspect of the final state. This gives a "difference" state whose tags indicate bugs in the code.

The difference state has one tag, corresponding to an omission of the dependency

mark(b) on mark(b)

This dependency was implicit in the specification because *mark(b)* appeared on the left-hand side of no assertion and was therefore taken to be invariant. If the new mark is equal to the old mark, it must depend on it.

Incidentally, we have here another example of the specification's expressing information not derivable from the code. The final state not only has missing dependencies, but extra dependencies too: *cursor(b)* on *text(b)* and *c*. These are syntactic and not semantic dependencies, by the following argument. The *search* operation looks forward and not backwards, and so *k* must represent a position ahead of the cursor. Executing *cut_region* affects the cursor only when the mark is behind it; but in this case, it is guaranteed to be ahead of it, and so the cursor is invariant.

This bug is a specification error; we forgot to consider the mark. If the mark follows the zap character, we probably want it to move backwards in

[3]A preview for the impatient: alias analysis can be smoothly integrated (Chapter 3) and relationships between aspects can be represented in the tag sets (Section 8.1). Tagging is also easily adapted to richer analyses, such as program slicing, in which the tags of an aspect can represent program lines in which that aspect is modified.

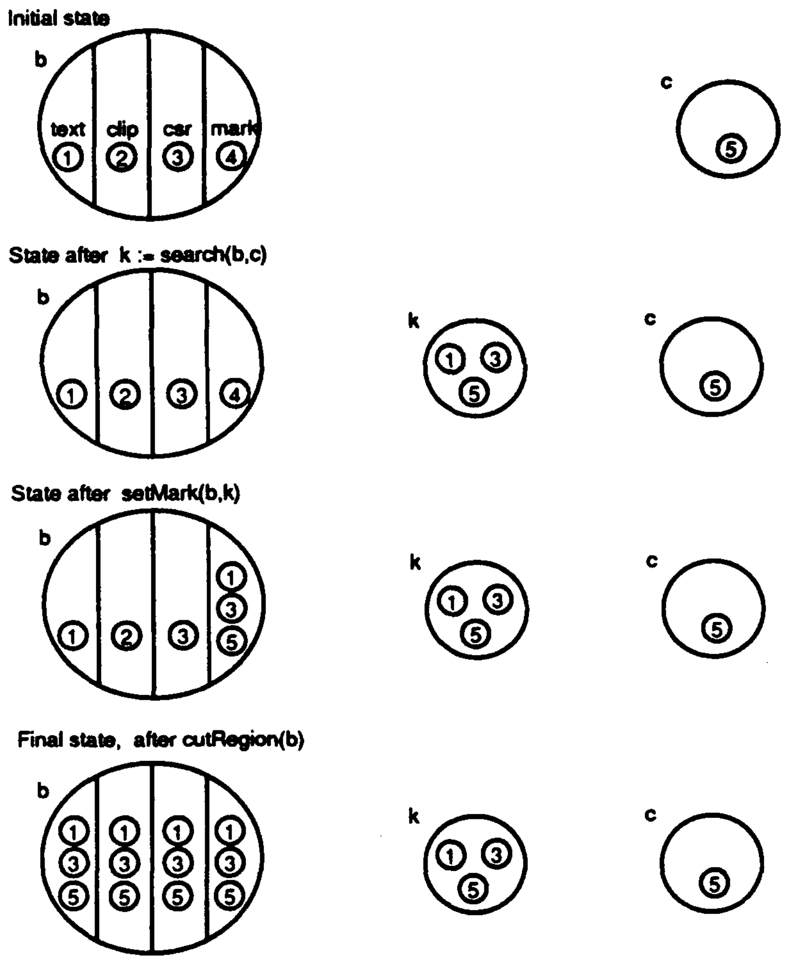


Figure 2.6: The calculation of dependency states for the code of *zap*

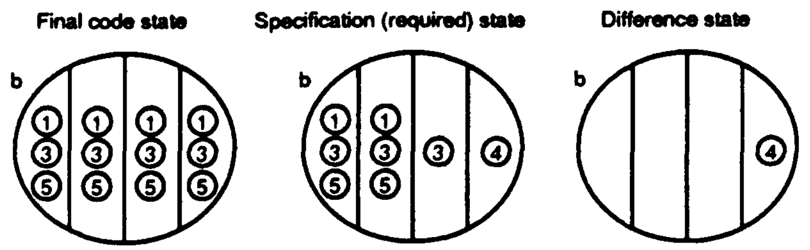


Figure 2.7: The comparison of the final state and required state

```
zap = proc (b: buf, c: char)
    %@ clip(b), text(b) ← text(b), cursor(b), c
    %@ mark(b) ← cursor(b)
    k: int := buf$search (b, c)                              1
    buf$set_mark (b, k)                                      2
    buf$cut_region (b)                                       3
    set_mark_at_cursor (b)                                   4
    end zap
```

Error	Missing Dependencies
swap lines 2 & 3	*clip(b), text(b)* ← *c*
swap lines 3 & 4	*clip(b), text(b)* ← *c*
omit line 3	*clip(b), text(b)* ← *text(b), cursor(b), c*
omit line 4	*clip(b), text(b)* ← *text(b), cursor(b), c*
del_region/cut_region	*clip(b)* ← *text(b), cursor(b), c*

Figure 2.8: Some bugs detected in *zap*

the buffer with its associated character. If it falls within the zapped text, it is not at all clear where it should go. Perhaps the simplest solution would be always to set the mark to the cursor position:

mark(b) ← *cursor(b)*

2.10 A Colony of Bugs

The corrected version of *zap* is shown with its new specification in Figure 2.8. The table shows the messages that would be generated by the checker for some faulty variants of the code.

Three classes of error are illustrated: omissions, misorderings and substitutions. Since the essence of the Aspect scheme is to find missing dependencies, omissions are particularly likely to be caught; in fact, omission of any line of *zap* is detected. But other kinds of error often lead to missing

```
zap_local = proc (b: buf, c: char)
    %@ clip(b), text(b), mark(b) ← text(b), cursor(b), mark(b), c
    %@ clip(b) ← clip(b)
    k: int := buf$search (b, c)
    m: int := buf$get_mark (b)
    if k < m then
          buf$set_mark (b, k)
          buf$cut_region (b)
          set_mark_at_cursor (b)
          end
    end zap_local
```

Figure 2.9: Aspect control-flow dependencies

dependencies too.

The errors shown here would not have been detected by existing static analysis schemes. A dataflow anomaly detector that finds misorderings like references to uninitialized variables could not identify the ones here, nor could a type checker catch the substitution errors. The Aspect checker finds them all quickly without user intervention.

2.11 Control Dependencies

Aspect dependencies do not only arise from the flow of data. Figure 2.9 shows an example of a procedure with an aspect dependency caused by the branching of an if-statement.

The function of *zap_local (b, c)* is to zap to the first occurrence of the character *c* only if it lies before the mark; otherwise it does nothing. Its specification differs from that of *zap* (Figure 2.8) by having the extra dependency assertions

text(b), clip(b) ← mark(b)
mark(b) ← mark(b), text(b), c
clip(b) ← clip(b)

The source of the aspect dependencies is not reflected in the form of the assertion; no caller of the procedure can observe whether a dependency is a flow dependency or a control dependency.

How are these assertions satisfied? The checker adds control dependencies of every aspect modified in the body of the if-statement on the result of the if-test. The modified aspects—*clip(b), text(b), mark(b)*—come from the specifications of the called operations: they are all the aspects that are not invariant because they appear on the left-hand side of an assertion. The outcome of the conditional expression depends on k and m (from the specification of $<$) which in turn depend on *text(b)*, *cursor(b)*, *mark(b)* and c in the pre-state (from the specifications of *search* and *get_mark*).

If-statements raise another issue in the dependency calculation. Since there are two paths an execution may take through the procedure, there are two possible final dependency states. The checker must account for dependencies due to either path, so it forms a single state that merges the dependency sets of the aspects of the two states. In general, this over-estimates the dependencies since it includes paths that can never occur, but since an error message is generated only for missing dependencies it never causes spurious reports.

Loops are handled in a similar way. The conditional test introduces control dependencies, and the resulting dependencies of all possible paths have to be merged. Although there are an infinite number of paths, there are a finite number of aspect dependencies, so the checker terminates when merging the dependencies of the resulting state of an iteration adds nothing new. In the worst case, the number of iterations is the square of the number of aspects (since each iteration adds a dependency, and each aspect can depend at most on every other aspect). In practice, though, the evaluation of loops terminates after two or three iterations.

2.12 Specifying and Checking Exceptions

CLU, like some other modern programming languages, provides an exception mechanism. The procedure of Figure 2.10 is like *zap_local*, but instead of doing nothing when the character lies beyond the mark, it signals *cannot*.

The two cases corresponding to the branches of the if-statement are not merged into a single state this time, because which branch is taken is ob-

```
zap_local2 = proc (b: buf, c: char) signals (cannot)
    %@ clip(b), text(b) ← text(b), cursor(b), c
    %@ mark(b) ← cursor(b)
    %@ except cannot: mark(b) ← cursor(b)
    %@ depending on cursor(b), mark(b), text(b), c
    k: int := buf$search (b, c)
    m: int := buf$get_mark (b)
    if k < m then
        buf$set_mark (b, k)
        buf$cut_region (b)
        set_mark_at_cursor (b)
    else
        signal cannot
        end
    end zap_local2
```

Figure 2.10: Specifying exceptional dependencies

servable in the calling context of the procedure. Instead, the specification separates the dependencies for the two ways in which the procedure can return, and the checker assesses them individually. The aspects that determine whether a signal is raised must also be specified (in the *depending on* clause).

The assertion for the signal case claims that *mark(b)* should depend on *cursor(b)*; this dependency arises only for the normal case, so an error message

Missing for signal 'cannot': mark(b) on cursor(b)

is displayed.

Exceptions are a boon to Aspect because they factor distinctions in the outcome of a procedure into the control flow, where they are amenable to static analysis. If a procedure is not structured into exceptional cases, the dependencies must be grouped together and the checker cannot be so discriminating.

To check code that calls procedures that raise exceptions, the checker must maintain a separate dependency state for each flow of control. These are handled like the results of the branches of an if-statement, but their merging is trickier. Each exceptional state is matched to its appropriate handler; the resulting states after the exceptions have all been handled can then be merged together.

Consider this calling context for example:

```
zap_local2 (b, c)
    except when cannot:
        failed := true
        buf$del_region (b)
        end
```

If *c* is beyond the mark, *zap_local2* signals *cannot*, control passes to the handler, the variable *failed* is set to true and the region is deleted. If *c* precedes the mark, the usual zapping occurs and the handler is not executed. To construct the dependencies of this code, the checker obtains two states from *zap_local2*'s specification. It evaluates the handler from the exceptional state; the other state (for the normal case) is merged with the state at the end of the handler.

Whether *zap_local2* signals or returns normally also gives a control dependency in this calling context. The *depending* clause of the specification therefore says which aspects determine the kind of termination, and, like the aspects governing the outcome of an if-statement's condition, these give further dependencies of the aspects modified in the handler. The value aspect of *failed*, for instance, will depend on *cursor(b)*, *mark(b)*, *text(b)* and *c*.

2.13 Summary

In this chapter, we have seen how a specification of aspect dependencies can be used to find bugs in a procedure. Checking involves the calculation of a maximal set of dependencies; these are compared to the minimal requirements of the specification and any missing dependency is taken to be evidence of a bug.

This only works because of the notion of an aspect. We would find few bugs as missing dependencies otherwise and these would usually be anomalies: not many procedures have results that do not depend in some way on all their arguments. Dividing an object into components enriches the dependency structure enough to catch interesting bugs.

A specification of a procedure plays two roles: as a criterion for checking the procedure's code and as a replacement for the procedure in checking its callers. We saw how a specification is more than a summary; the specifier can say that a dependency is missing (and is to be taken as accidental if nonetheless it appears in the procedure's code). This allows more bugs to be caught.

An aspect is just a name for a component of an object. The choice of aspects is left to the programmer or specifier. The more aspects, the more bugs can be detected, but the more complicated the procedure specifications become. The lack of interpretations for aspects makes specifications easier to write than conventional specifications and makes automatic checking possible.

The soundness of this scheme rests on two assumptions:

1. The dependencies calculated by the checker include at least all the dependencies that actually arise. In the absence of aliasing, this is a property of the dependency algorithm we have described. Chapter 3 describes the more complicated construction necessary to deal with aliasing.

2. The aspects of a type are independent. Aspects are more like projections than components; they can in fact overlap, but we assumed so far that one cannot subsume another. If the *cursor* aspect could be derived from the *mark* aspect, a requirement to depend on *cursor(b)* could be discharged with a dependence on *mark(b)*. This restriction is relaxed in Chapter 8.

The parts of Aspect presented so far are not tailored to CLU. The notion of aspects and aspect dependencies could be applied equally to LISP or Fortran. Once we consider the structure of the store, fine differences in the semantics appear and this is no longer true.

Chapter 3

Reference Aspects

In the last chapter, we saw how to catch bugs as missing dependencies between aspects. Although we talked of dividing an *object* into aspects, we could equally well have have regarded aspects as components of *variables*. We might then have explained *cursor(b)*, for example, not as the cursor aspect of the buffer object called b, but rather the cursor aspect of the variable b, and the analysis would have been the same.

Not all objects, however, can be named by program variables. The first step we take in this chapter is to introduce a general naming scheme for objects. By labelling the references between objects, we can name objects arbitrarily deep in the program state. For example, if x is a variable that names an object O_1 and O_1 has a reference r to another object O_2, then we give O_2 the name $x.r$; if O_2 has a reference s to O_3, then O_3 is called $x.r.s$, and so on. These references are just another kind of aspect called *reference aspects.* They come in two varieties: *pointer aspects*, for references to single objects, and *collection aspects*, for references to sets of objects (like the elements of an array).

Because of aliasing, we cannot determine dependencies reliably unless we know what references there may be between objects. Specifications must give *values* to reference aspects so that the set of possible aliases can be bounded. This is expressed with *binding assertions.* The result is a more complicated dependency calculation, in which the structures of objects, as well as their dependencies, evolve.

Finally, there are *allocation assertions* to describe dynamic allocation: in CLU, new objects may be created at runtime and existing objects, like arrays, may grow.

3.1 Pointer Aspects

So far we have only considered the aspects of objects named by program variables. We found some bugs in *zap* with a specification on buffer aspects:

```
zap = proc (b: buf, c: char)
    %@ clip(b), text(b) ← text(b), cursor(b), c
```

Suppose now that we have an object *w* of an abstract type *window* that contains references to two buffer objects, an upper buffer and a lower buffer, and we want to specify the procedure

```
zap_in_upper = proc (w: window, c: char)
```

whose effect is a zap to character *c* in the upper buffer of *w*. How can we say this? We are stuck, because we have no way to talk about the zapped buffer, the buffer objects having no names in the procedure header.

A *pointer aspect* is a special kind of aspect that denotes a reference to another object. In the same way that we declared *cursor* to be an aspect of *buf*, we can declare the *window* type to have two pointer aspects, *upper* and *lower*:

```
window = cluster is ...
    %@ aspects *upper: buf, *lower: buf
```

The stars mark the aspects as pointers. Also, the pointer aspects, unlike the plain aspects of *buf* (such as *cursor* and *text*) have types. This specification now allows us to form object names. Using the window variable *w*, we can denote a buffer reached from *w* by *w.upper*, and refer to the aspects of that buffer as *cursor(w.upper)*, *text(w.upper)*, etc.

In general, an object name is a variable followed by a sequence of pointer aspects. For example, our editor may have a stack of windows *s*, and the stack specification may define a pointer aspect *top* for the top element of the stack. We can then write *s.top* for the top window of the stack, and *s.top.upper* for the upper buffer of that window.

Terms like *cursor(w.upper)* are called *aspect expressions*. An aspect expression is formed from an aspect and an object name[1]. With this ability

[1]We shall see later that pointer aspects may also be applied to object names to form aspect expressions like *upper(w)*. The meaning of such an expression is the *value* of the pointer. The distinction between *upper(w)*, which is a property of the object *w*, and *w.upper*, which is a name for a buffer object, is important and explained in detail later.

to name aspects of objects beyond those attached to variables, we can now write our specification:

```
zap_in_upper = proc (w: window, c: char)
    %@ clip(w.upper), text(w.upper) ←
    %@    text(w.upper), cursor(w.upper), c
```

and we move on to the question of how such a specification is checked.

3.2 The Problem of Aliasing

When aliasing can happen, the dependency calculation of Chapter 2 does not work: it misses dependencies that are present. Aliasing lets dependencies sneak in unnoticed, because a statement that appears to change some variable may change other variables whose names do not appear in it. This section shows how aliasing arises even in a reasonable implementation of *zap_in_upper.*

Suppose the window cluster provides a procedure *get_upper* that, given a window object, returns the upper buffer of the window. A bad specification of *get_upper* is shown in Figure 3.1; it claims that each aspect of the resulting buffer depends on the corresponding aspect of the upper buffer of *w*. We will not be able to write the correct specification until Section 3.3. The code of *zap_in_upper*, on the other hand, is fine. Its intention is that *b* become an alias for the upper buffer of *w*, so that *zap*'s change to *b* is a change to the upper buffer too. The assignment statement does not make a copy of the buffer but makes *b* a name for the object returned by *get_upper*.

Running the checker on *zap_in_upper* with the specifications of Figure 3.1 will generate a flurry of messages such as

```
Missing: clip(w.upper) on text(w.upper)
```

showing, in fact, all the required dependencies to be missing. But the code is nonetheless correct. Why then are these dependencies not found?

The dependency of an aspect like *text(w.upper)* should arise from the execution of *zap*. But the dependency calculation will miss this, since *w* and its components will be assumed to be invariant over the call to *zap(b,c)*.

```
zap_in_upper = proc (w: window, c: char)
    %@ clip(w.upper), text(w.upper) ←
    %@   text(w.upper), cursor(w.upper), c
    b: buf := window$get_upper (w)
    zap (b,c)
    end zap_in_upper

get_upper = proc (w: window) returns (buf)
    %@ text(result) ← text(w.upper)
    %@ clip(result) ← clip(w.upper)
    %@ cursor(result) ← cursor(w.upper)
    %@ mark(result) ← mark(w.upper)
```

Figure 3.1: A failure to specify aliasing in a called procedure

3.3 Binding Assertions

To solve this dilemma, the analysis of the code must take into account the aliases that might hold at any point. The dependencies that are constructed will then be determined in part by the aliasing. If a statement causes some object x to acquire a dependency, and y is an object aliased to x, then y must acquire that dependency as well.

Since a procedure can establish an aliasing, we need to be able to specify this effect. Since the checker does account for aliasing, the messages it generates for *zap_in_upper* are due not to a deficiency in its mechanism but rather to the specification of *get_upper* in Figure 3.1. That specification says that the aspects of the resulting buffer depend on their counterparts in *w.upper*, but this is not enough, since it allows implementations that return a copy of *w.upper*. We need to say that the buffer returned *is w.upper*. The correct specification

```
get_upper = proc (w: window) returns (buf)
    %@ result() :– upper(w)
```

uses a ***binding assertion*** to say that the object named by the result is the object that was initially named by the *upper* pointer aspect of *w*.

So far we have insisted that aspects have no values; we only traced dependencies between aspects. Aliasing forces a compromise. To say that two objects are aliased is equivalent to saying that the pointers to those objects (which may be variables or pointer aspects) have the same value. So a binding assertion defines the *values* of aspects of the post-state in terms of the values of the aspects of the pre-state. The specification of *get_upper* can be read as: 'the value of the pointer *result* in the post-state is the value of the *upper* pointer aspect of w in the pre-state'. In general, the binding assertion α :– β says that there is an execution in which the value of the aspect expression α in the post-state is equal to the value of β in the pre-state.

The pointer aspect *upper* thus has the signature

$$upper : window \longrightarrow buf\text{-}pointer$$

in contrast to a plain aspect like *mark* whose signature is

$$mark : buf \longrightarrow AspectRange$$

The distinction is purely pragmatic. It may well be a useful elaboration of Aspect to allow specifications that give values to plain aspects[2], but the dependency scheme will work without it. Giving values to pointer aspects, on the other hand, is crucial, because without it aliasing could not be described and spurious bugs would be reported.

A final comment on the syntax of the binding assertion

result() :– *upper(w)*

is in order. The assertion *a(p)* :– *b(q)* always describes a change to the a aspect of the object called p. The only change caused by *get_upper* is to the environment which maps variable names to objects. It is convenient to regard the variable names to be pointer aspects of the environment, and to regard the environment as an object whose name is just the empty string. Thus the aspect expression *result()* denotes the *result* aspect of the environment, and its occurrence on the left-hand side of the binding assertion indicates a change to the environment itself.

The aspect expression *upper(w)* denotes the value of the pointer aspect of the object w. This is to be distinguished from *w.upper*, which is an object

[2]This is discussed in Section 8.6.

name for the buffer reached by dereferencing that pointer. The value of *upper(w)* can also be set by a binding assertion, such as in the specification of *window$set_upper(w, b)* which binds the upper buffer of *w* to *b*:

```
set_upper = proc (w: window, b: buf)
    %@ upper(w) :— b()
```

Here it is clear why *upper(w)* rather than *w.upper* is written on the left-hand side: the object that is modified is *w* and not the buffer *w.upper*[3].

3.4 Graph Evaluation

Now that pointer aspects are given values by binding assertions, the checker has to keep track of them. A new state is needed—a graph will do. Figure 3.2 shows the initial state of *zap_in_upper*. Each node (oval) represents an object; the rectangle at the root of the graph is the environment that maps the formals of *zap_in_upper* to objects. The objects are divided into aspects in the same way as before, but now each aspect may have both dependences and values. The values are recorded below the horizontal line dividing the box. For the plain aspects, the values are not known, so they are written as question marks. The values of the pointer aspects are locations (or identities) of other objects and are recorded as edges of the graph.

Starting at the top, we see that the variables (pointer aspects of the environment) have values that are the locations of two objects: *c* points to a character object with a single (unnamed) aspect, and *w* to a window object. The window object has two pointer aspects *upper* and *lower*, whose values are the locations of two buffer objects, each of which is divided into four plain aspects.

To execute the first statement

```
b: buf := window$get_upper (w)
```

[3]The reader may still wonder why *upper(w)* need be written on the right-hand side in the specification of *get_upper*, instead of defining the syntax so that *result()* :— *w.upper* means 'make the result point to the object called *w.upper*. The reason is simply to avoid the L/R-value complication of programming languages. An assertion that says that the upper buffer is unchanged would then have to be written awkwardly as

```
upper(w) :— w.upper
```

instead of *upper(w)* :— *upper(w)*.

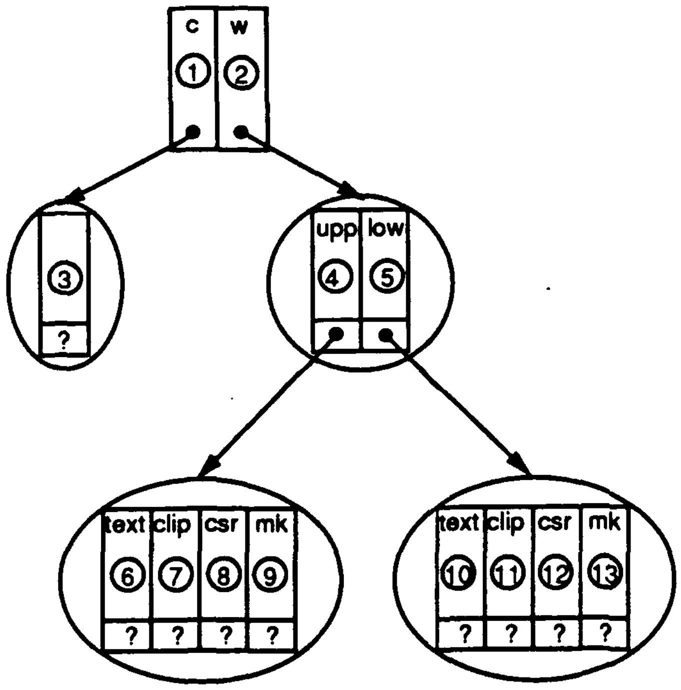


Figure 3.2: The initial state of *zap_in_upper*

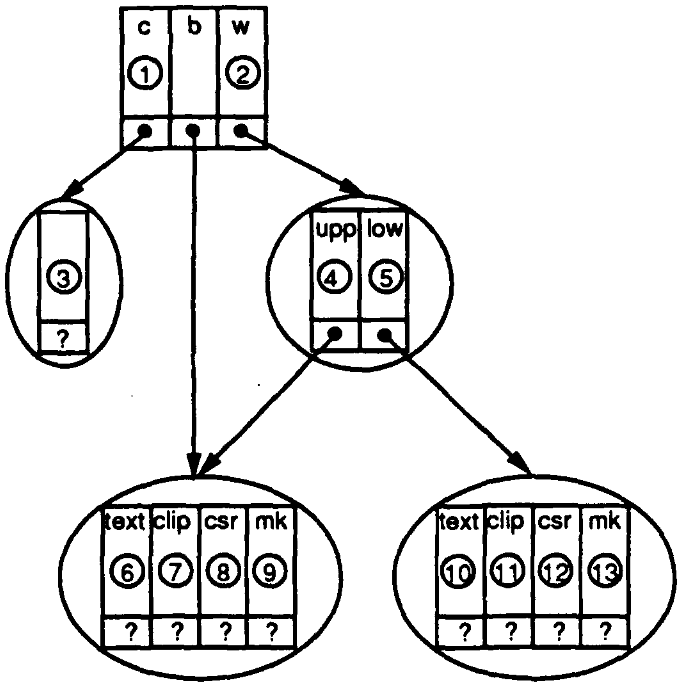


Figure 3.3: The state following *b: buf := window$get_upper (w)*

```
zap = proc (b: buf, c: char)
    %@ clip(b), text(b) ← text(b), cursor(b), c
    %@ mark(b) ← cursor(b)
```

Figure 3.4: The specification of *zap*

the checker looks up the specification of *get_upper*, which contains the single binding assertion *result() :— upper(w)*, and executes it by extending the environment with the variable *b* and then setting *b*()'s value to the location denoted by *upper(w)*. The resulting state (Figure 3.3) has two edges incident on one of the buffer objects. This is aliasing: the buffer has two names (*w.upper* and *b*), and a change under one will appear to be a change under the other too.

This happens in *zap_in_upper*'s next statement, *zap(b,c)*. The checker executes the dependency assertions of *zap* procedure's specification (repeated in Figure 3.4) as described in Chapter 2. Since the object names have been bound dynamically, the first step in finding the appropriate tags is to locate the objects. For example, to set the new tags for *clip(b)* from the assertion

clip(b) ← text(b), cursor(b), c

the checker first locates the objects *b* and *c* by looking up the variables in the environment (i.e., following the arcs from the root). Each aspect expression on the right yields a tag and these are written into the dependency box of the *clip* aspect of the object *b*, replacing the previous tag[4].

This gives the final state (Figure 3.5), which is checked against the specification

```
zap_in_upper = proc (w: window, c: char)
    %@ clip(w.upper), text(w.upper) ←
    %@    text(w.upper), cursor(w.upper), c
```

by looking up the dependences of the aspects of the object *w.upper*. For example, *clip(w.upper)* contains the three tags numbered *3*, *6* and *8*, which include the initial tags of the three aspects *text(w.upper)*, *cursor(w.upper)* and *c* obtained from the initial state (Figure 3.2).

[4]For reasons that are discussed in Chapter 4, the checker includes other tags too.

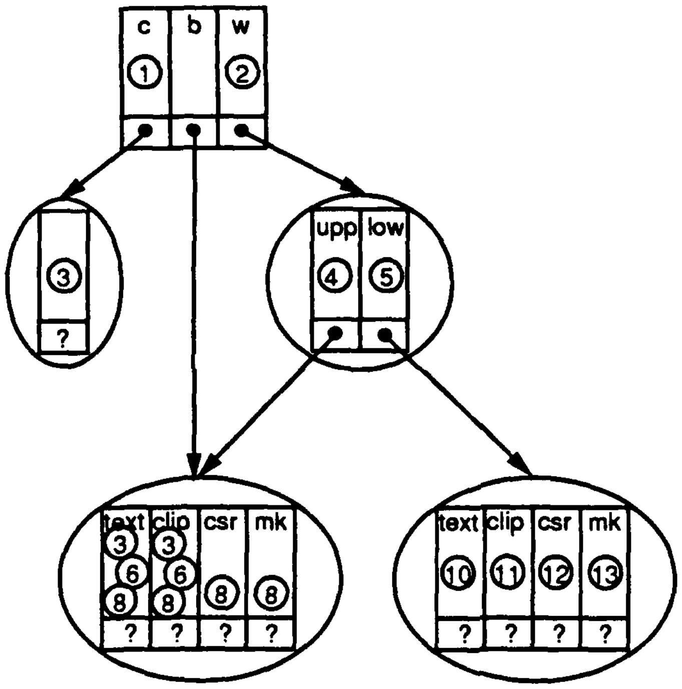


Figure 3.5: The final state of *zap_in_upper*

```
swap = proc (w: window)
    %@ upper(w) :– lower(w)
    %@ lower(w) :– upper(w)
    u: buffer := window$get_upper (w)
    window$set_lower (w, u)
    window$set_upper (w, window$get_lower (w))
    end swap
```

Figure 3.6: Checking a binding assertion

3.5 Checking Binding Assertions

We have only seen binding assertions in the specifications of called procedures (such as *get_upper*). Recall, though, that all specifications play two roles: not only do they provide information about a procedure for checking its callers, but they also give the criteria for checking the procedure itself.

Consider the procedure *swap* which is intended to swap the upper and lower buffers of a window. Its specification and a candidate implementation are given in Figure 3.6. The checker evaluates the code and then examines the values of the *upper* and *lower* pointers of *w* in the final state. The value of *upper* is required by the specification to be the value that *lower* had in the initial state, and vice versa.

This test is analogous to the dependency test. There, to test the assertion $\alpha \leftarrow \beta$, we checked that the final tags of the aspect α's dependences were a superset of the initial tags of the aspect β. Here, to test the assertion α:– β, we check that the possible final values of the pointer aspect α are a superset of the initial value of β.

The *swap* procedure does not satisfy its first assertion, and the checker would display

```
Missing: upper(w) :– lower(w)
```

3.6 Collection Aspects

The aspects of an object are determined by its type. They are a fixed set, the same for every object of the type[5]. The pointer aspects are no different from the rest, and can thus only refer to a fixed number of objects. This raises a problem for objects like sets, arrays and lists that can grow as the program executes. It is not acceptable to provide a large but fixed number of pointer aspects; even arrays in CLU can grow without bounds.

A *collection aspect* is a reference aspect that denotes a set of references. This set is not bounded, so a single collection aspect may represent any number of references to other objects.

A stack for example might have three aspects (Figure 3.7). The first, *size*, is a plain aspect denoting the number of elements in the stack. The second, *top*, is a pointer aspect that represents the location of the top element of the stack. The third, *rest*, is a collection aspect representing the set of locations of the remaining elements. The classification of the aspects is indicated by stars: none for a plain aspect, one for a pointer and two for a collection.

Let us examine the operations in turn. The *new* operation allocates a new stack and returns it; it is discussed in Section 3.8. The first assertion of *push* places the element *e* at the top of the stack. The third assertion just says that the size of the stack after a *push* depends on its size before.

The second assertion is more interesting. It says that the set of objects that are below the top of the stack afterwards include whatever was below before, plus the element that was previously on the top. The two expressions on the right-hand side can be thought of as defining two possibilities for an element that is in *rest(s)* after: either it came from an element in *rest(s)* before, or it was on the top of the stack. In general, a binding assertion gives a set of possibilities.

The *pop* operation is similar to *push*. The top element is returned by

[5]In explaining the syntax of binding assertions such as

result () :– *b*

we said that the program variables are regarded as pointer aspects of the environment, which could be viewed as an object. The number of variables in scope may grow and shrink, however. The analogy of variables as aspects does not stretch this far; its only point was to motivate the syntax. The formal semantics treats the environment as a distinct entity unlike any object in the store.

```
stack = cluster [t: type] is new, push, pop, top, size

    %@ aspects
    %@    size,                 % number of elements in stack
    %@    *top: t,              % the top element
    %@    **rest: t             % the remaining elements

    new = proc () returns (stack[t])
        %@ s: stack[t]
        %@ result() :– s()

    push = proc (s: stack[t], e: t)
        %@ top(s) :– e()
        %@ rest(s) :– rest(s), top(s)
        %@ size(s) ← size(s)

    pop = proc (s: stack[t]) returns (t)
        %@ result() :– top(s)
        %@ top(s), rest(s) :– rest(s)
        %@ size(s) ← size(s)

    top = proc (s: stack[t]) returns (t)
        %@ result() :– top(s)

    size = proc (s: stack[t]) returns (int)
        %@ result ← size(s)
```

Figure 3.7: Specification of a stack

```
switch = proc (w: window, s: stack[buf])
    %@ upper(w) :– top(s), upper(w)
    %@ top(s) :– upper(w), top(s)
    if stack[buf]$size (s) = 0
        then return
    else
        u: buf := window$get_upper (w)
        t: buf := stack[buf]$pop (s)
        stack[buf]$push (s, u)
        window$set_upper (w, t)
        end
    end switch
```

Figure 3.8: A procedure that uses a stack

the first assertion. The second assertion is short for two elementary binding assertions:

top(s) :– *rest(s)*
rest(s) :– *rest(s)*

The first says that the top element after a *pop* was one of the elements below the top before[6]; the second says that the elements that are below the top after include those that were below the top before. The indeterminacy arises because the collection aspects denote sets and not sequences. The order of the elements below the top of the stack is not expressible in Aspect—unable to say that *top(s)* becomes the first element in *rest(s)*, we are left saying that it becomes one of the elements of *rest(s)*.

The *top* operation is like *pop*, but it leaves the stack invariant (evident from the omission of any assertion with a stack aspect on the left-hand side) and returns the top element. Finally, the *size* operation returns an integer that depends on the size of the stack.

The *switch* procedure (Figure 3.8) takes a window and a stack of buffers, and switches the top buffer of the stack with the upper buffer of the win-

[6]This is slightly simplified. The specification assumes that *pop* is not called on an empty stack, in which case *top(s)* :– *top(s)* would be needed as well.

dow. Its specification, like that of *push*, gives several possibilities. The first assertion, for example, says that the upper buffer after is either what was the upper buffer before (if the stack is empty), or what was the top buffer of the stack.

All the given possibilities must occur in the code. In the same way that the checker lists missing dependencies, it will list any missing bindings. For example, if we forgot to handle the case of the empty stack (omitting the 'if' and leaving only the code following the 'else'), there would be no path in which, after the execution, *top(s)* has the value it had before, and an error message would be generated. Another bug would be to swap the push and pop statements, giving

Missing: upper(w) :— top(s).

3.7 Polymorphism

The stack specification did not specify the type of the element objects. Instead, the aspect declarations use *t* as the type of the object that *top* and *rest* point to. This is not a type but a parameter of the *stack* cluster. The cluster can be instantiated with any type for *t*, so that the same code (and specification) can be used *polymorphically* for a stack of integers or a stack of buffers.

In CLU, polymorphic types can only be instantiated at compile-time, so the Aspect specification can be viewed as a template in which *t* is replaced by the appropriate type. The checker constructs the object structure for *stack[buf]* from the aspect declarations of the two clusters, *stack* and *buf*.

The procedure specifications of *stack* need no special treatment, since the meaning of a binding assertion is unaffected by the type of the reference aspects it relates. Furthermore, there are no assertions about aspects of the element type[7].

3.8 Allocation Assertions

Binding assertions describe changes in the shape of the program state due to rearrangements of the objects. Some procedures change the state not only by

[7]This is a limitation of Aspect discussed in Section 8.3.

```
copy_to_upper = proc (w: window, b: buf)
    %@ b2: buf
    %@ text(b2) ← text(b)
    %@ upper(w) :– b2()
```

Figure 3.9: Assigning dependences to allocated objects

rearranging the existing objects, but also by allocating fresh objects which are then bound to existing objects or returned as results.

The *stack$new* operation (Figure 3.7) has an *allocation assertion* to say that a new stack object is allocated and a binding to associate it with the result variable:

s: stack[t]; result() :– s()

The assertion *x: t* declares *x* to be a freshly allocated object of type *t*. The name *x* is local to the scope of the specification and must not be the name of an argument of the procedure.

The *copy_to_upper* procedure (Figure 3.9) shows how an allocated object may become part of an argument object[8]. It replaces the upper buffer of the window *w* with a new buffer object. The aspects of the allocated object may be given dependences; the *text* of the new buffer comes from the buffer *b*.

3.9 Immutable Objects

An object in CLU is either mutable, which means that it can change over its lifetime, or immutable, which means that once created its value is set forever. Assignment of immutables still introduces sharing, but the effect is not visible since no mutations of the shared object are possible. So aliasing is never a problem.

Buffers are mutable, because they have operations like *buf$cut_region(b)*. Integers are immutable: there are no operations that change their value. The statement

[8]It may not be bound to an argument variable because CLU semantics prevents a procedure call from mutating the environment, except in assigning the result.

```
x := x + 1
```

does not mutate x, but makes x a name for the new integer object whose value is one greater than the old integer object named x. But whether this happens or the value of x is increased by one is a question of viewpoint, since the distinction is not observable.

Aspect, like CLU, gives assignment the same semantics for mutable and immutable objects. The integer add operation allocates a fresh integer object:

```
add = proc (i, j: int) returns (int)
    %@ k: int
    %@ k ← i, j
    %@ result() :– k()
```

This is a frequent idiom, so it can be abbreviated to

```
add = proc (i, j: int) returns (int)
    %@ result ← i, j
```

We have already seen several examples of this, such as *getMark* (Figure 2.1).

3.10 Summary

We started with a naming scheme for structuring the program state. The notion of an aspect as a property of an object extends naturally to references within that object to other objects. Two kinds of reference aspect were introduced: pointer aspects, for references to single objects, and collection aspects for references to sets of objects. Like plain aspects, the reference aspects of a type are chosen by the programmer. For a stack, we chose to represent the elements with two aspects, a pointer aspect for the top element and a collection aspect for the rest.

Aliasing foils the dependency calculation of Chapter 2 because it gives an invisible route for side-effect dependencies. To account for this, we added binding assertions to give the values of reference aspects and we modified the dependency calculation by representing possible references between objects as the edges of a graph.

Dynamic allocation called for a third kind of assertion. The functional parts of CLU can be handled in the same framework as before by viewing immutable objects as allocated on the heap.

Chapter 4

Refinements

Chapter 2 explained dependency assertions and plain aspects. Then Chapter 3 introduced reference aspects to name objects that are not immediately associated with variables, and showed how binding assertions describe changes in the shape of the store: how objects are connected and, in particular, when two names refer to the same object.

This chapter brings these ideas together. The notion of references forces us to revisit dependency assertions and say more precisely what they mean. The happy outcome of this complication is more expressive power. We shall see that the checker can catch some bugs that could not be explained before and also how the checker can give more informative messages by distinguishing the multiple claims of a single assertion.

4.1 Reference Dependencies

So far we have seen that plain aspects have dependencies and reference aspects have values. We needed the values of reference aspects to determine possible aliasings, and thus account for extra dependencies that creep in through side-effects. This is a pragmatic distinction; plain aspects could be given values too[1] but they are not essential.

The question that arises is: do reference aspects have dependencies as well as values? The answer is yes; not only do they, but they must. In fact, dependencies on reference aspects are implicit in most of the assertions we have already looked at.

[1]See Section 8.6.

Our starting point is the dependency assertion

$$a(p) \leftarrow b(q)$$

Up until now, we have read the arrow as "depends on". This will no longer do. In the next section, we shall see that this assertion implies other dependencies and some new vocabulary is therefore needed. So we shall henceforth read $\leftarrow$ as "affected by", and "$a(p)$ is affected by $b(q)$" will mean that:

1. the aspect b of the object Q is read;
2. the aspect a, of the object P is written; and
3. the post-state value of the a aspect of P depends on the pre-state value of the b aspect of Q; where
4. P is the object named by p in the pre-state, and
5. Q is the object named by q in the pre-state.

There are two new points here. First, all object names in aspect expressions name objects in the pre-state. Second, there are extra implicit reference dependencies:

- because of (1), $a(p)$ depends on the reference aspects that made q name Q—that is, if q is $y.f_1.f_2..f_n$, then $a(p)$ depends on $f_n(y.f_1..f_{n-1})$, $f_{n-1}(y.f_1..f_{n-2})$, as far as $y()$; and
- because of (2), $a(p)$ depends on the reference aspects that made p name P—that is, if p is $x.e_1.e_2..e_n$, then $a(p)$ depends on $e_n(x.e_1..e_{n-1})$, $e_{n-1}(x.e_1..e_{n-2})$, as far as $x()$.

There is an implicit dependency assertion in every binding too. A binding assertion α:– β means that α (after) is made equal to β (before), so it must also be affected by β. Thus α :– β includes $\alpha \leftarrow \beta$ implicitly[2].

The next few sections explain why assertions have these interpretations. We shall start with the notion of reference dependencies and see why the implicit claims are natural. Then we shall look at how Aspect binding assertions define a set of possibilities; how objects are named always in the pre-state; and finally, how implicit dependencies allow more bugs to be caught.

[2]Unless $\alpha = \beta$ and there are no other binding assertions, in which case α is invariant and no dependency assertion is implied.

```
replace = proc (w: window, b: buf, cmd: string)
    %@ upper(w) :– b(), upper(w)
    %@ lower(w) :– b(), lower(w)
    %@ lower(w), upper(w) ← cmd
    if cmd = 'upper' then
        window$setUpper (w, b)
    elseif cmd = 'lower' then
        window$setLower (w, b)
        end
    end replace
```

Figure 4.1: Illustration of pointer dependencies

4.2 Rationale for Reference Dependencies

Suppose we have a procedure *replace* (Figure 4.1) that takes a window *w*, a buffer *b* and a command *cmd*, and replaces the upper or lower buffer of *w* with *b*, according to whether *cmd* is 'upper' or 'lower'.

The initial value of the string *cmd* clearly affects the final state of the procedure. But where should the dependence on *cmd* be recorded? We certainly do not want to say that an aspect of either of the buffer objects is made to depend on *cmd*, for we know the buffers to be invariant. The change is to the *window* object: *cmd* determines the final values of the *upper* and *lower* pointers of *w*. Therefore it is the pointer aspects *upper(w)* and *lower(w)* which must have a dependence on *cmd*. These dependences may be specified, just like the dependences of plain aspects.

Reference aspect dependencies have more subtle repercussions than regular dependencies. Suppose we are constructing dependencies for:

```
replace (w, rb, cmd)
b: buf := window$get_upper (w)
c: int := buf$get_cursor (b)
```

The final value of *c* depends on the initial value of *cmd*, because, if *cmd* is different, *c* will be the cursor of a different buffer. This dependency is accounted for by implicit and explicit dependencies in the specifications of the operations. Here is how it arises, viewed from the first statement down:

- The assertion of *replace*

  ```
  upper(w) ← cmd
  ```

  gives an explicit dependence of *upper(w)* on *cmd*—nothing strange here.

- The assertion of *get_upper*

  ```
  get_upper = proc (w: window) returns (buf)
      %@ result() :– upper(w)
  ```

  implies a dependence of *result()* on *upper(w)*, since if *result()* acquires the value of *upper(w)*, it must surely depend on it. The variable *b* is bound to the result of *get_upper*, and so we now have a dependence of *b()* on *cmd*.

- The assertion of *get_cursor*

  ```
  get_cursor = proc (b: buf) returns (int)
      %@ result ← cursor(b)
  ```

  implies a dependence of *result* on *b()*. The outcome of reading the *cursor* aspect of *b* must depend on which object *b* is. To put it another way, reading *cursor(b)* involves reading *b()* first. Since *c* is bound to the result of *get_cursor*, this gives the dependence of *c* on *b()*, and hence on *cmd*.

The implicit dependency in *get_cursor* was due to the reading of *cursor(b)*. An analogous dependency arises when an aspect of an object is written. Suppose that, instead of obtaining the cursor in the above code fragment, we modify it:

```
replace (w, rb, cmd)
b: buf := window$get_upper (w)
buf$set_cursor (b, i)
```

Consider the final value of the *cursor(w.upper)*. By a similar argument, it must depend on *cmd*, because whether the call to *set_cursor* affects it depends on whether *w.upper* and *b* are aliased. This time, the dependency is due to *set_cursor*'s assertion

```
set_cursor = proc (b: buf, c: int)
    %@ cursor(b) ← c
```

which implies a dependence of *cursor(b)* on *b()*, which as before depends on *cmd*. This implicit dependency is more subtle. It arises because *cursor(b)* denotes the dependencies of the cursor of the *object* called *b*, not of the variable *b*. From the viewpoint of the object called *w.upper*, its cursor is modified only when the value of *b()* is its identity.

Collection aspects imply dependencies in exactly the same way as pointer aspects. Given a stack of buffers *s* (Figure 3.7), a procedure that returns the total size of all the text in all the buffers would have the assertion

```
result ← text(s.top), text(s.rest)
```

Recall that *rest* is a collection aspect and *top* is a pointer aspect, so that *s.top* denotes the buffer at the top of the stack and *s.rest* denotes a buffer below the top. This assertion implies that *result* depends on *top(s)*, *rest(s)* and *s()*: which buffer is the top of the stack (a pointer dependency), which buffers are in the rest of the stack (a collection dependency) and which stack *s* is bound to.

4.3 Assertions As Possibilities

In general, an Aspect assertion defines a possibility, and a specification defines a set of possibilities. The specification of *replace* (Figure 4.1) illustrates this. It gives two possible values for *upper(w)* and lower(w):

```
upper(w) :– b(), upper(w)
lower(w) :– b(), lower(w)
```

The possibilities of *upper(w)* and *lower(w)* are independent. They are not correlated in any way, so the specification does not express the notion that only one, or even at least one, of *upper(w)* or *lower(w)* is set to *b()* during a single execution.

4.4 Prenaming of Objects

A final point about the meaning of an Aspect procedure specification concerns the relationship between binding and dependency assertions. By always interpreting object names in the pre-state, we can ensure that assertions retain their declarative meaning and do not interact with one another in peculiar ways.

The *replaceReset* procedure is like *replace*, but as well as replacing the upper or lower buffer of *w* with *b*, it resets the cursor in the replaced buffer to its top:

```
replaceReset = proc (w: window, b: buffer, cmd: string)
    %@ upper(w) :– b(), upper(w)
    %@ lower(w) :– b(), lower(w)
    %@ lower(w), upper(w) ← cmd
    %@ cursor(w.upper), cursor(w.lower) ← cmd
```

Which buffer is meant by *w.upper* in the last assertion? After the execution of the procedure, *w.upper* might name a different buffer (the one initially called *b*). To resolve this dilemma, names of objects are always interpreted in the pre-state, wherever they occur in an Aspect specification. So *w.upper* and *w.lower* both refer to the old buffers of *w*, and the buffer called *b* in the pre-state is not affected. If instead we wanted to specify the resetting of the cursor of the replacing buffer, we would replace *w.upper* by *b* in the last line.

The reason for choosing to name objects in the pre-state rather than the post-state is that some objects may no longer have names in the post-state. The replaced buffer in this example is only accessible to variables in the calling context that were already aliased to it; by definition the 'replaced' buffer is the one no longer reachable by any argument of *replaceReset*.

This rule simplifies complicated restructurings of objects and ensures that the order in which we write the binding assertions does not matter. Suppose we declare the record types

```
pair = record [one, two: int]
ppair = record [one, two: pair]
```

and then write the specification:

```
tricky = proc (p: ppair)
    %@ one(p) :– two(p)
    %@ two(p) :– one(p)
    %@ one(p.one) :– two(p.one)
    %@ two(p.one) :– one(p.one)
```

This has the simple effect of swapping the objects of the pair originally called *p.one*, and also swapping the objects of the pair (of pairs) *p*. The name *p.one* on the left-hand side of the third assertion refers to the object called *p.one* initially, and not the object called *p.one* after the effect of the second assertion has been taken into account.

It is tempting to read the assertion

```
one(p.one) :– two(p.one)
```

as a claim that the object called *p.one.one* afterwards was called *p.one.two* before. But this is not true, since *p.one* names a different object in the pre- and post-states.

4.5 Catching Extra Bugs

Implicit dependencies allow more bugs to be caught because they strengthen the specification. Our refined interpretation of dependency assertions requires that an object be read and another written. Previously, if an object was to be written with some constant value, we could not detect any errors because the object would have no required dependencies. Now, however, we can see how the requirement that the object be written can be checked, using dependencies on reference aspects.

Consider an implementation of *reset_cursor* (Figure 2.1), a procedure that resets a buffer's cursor to the top:

```
reset_cursor = proc (b: buf)
    %@ cursor(b) ← ∅
    buf$set_cursor (b, 0)
    end reset_cursor
```

Recall that ∅ denotes the empty list, so that the assertion says that the final value of the cursor is affected by no aspects of *b*. It does, however, imply a

```
exchange = proc (b: buf)
    %@ mark(b) ← cursor(b)
    %@ cursor(b) ← mark(b)
    c: int := buf$get_cursor (b)
    buf$set_mark (b, c)
    m: int := buf$get_mark (b)
    buf$set_cursor (b, m)
    end exchange
```

Figure 4.2: First example, revisited

dependence of *cursor(b)* on *b()*. Without the extra dependency, *SKIP* would satisfy the specification; with it, the buffer variable *b* must be dereferenced and a change made to that object.

Let us return, for a moment, to our very first example of a bug (Figure 4.2) taken from the start of Chapter 2. The reader may have wondered why the checker displayed the message

Missing: cursor(b) on mark(b)

when the assertion *cursor(b)* $\leftarrow$ *mark(b)* was not satisfied, instead of just

Missing: cursor(b) $\leftarrow$ *mark(b)*

The reason should now be clear. Even a simple dependency assertion implies several aspect dependencies. The checker determines which of these are missing and can give a more helpful message by listing them explicitly rather than giving only the violated assertion. If, in addition to the error of failing to save the initial value of the mark, we also omitted the *set_cursor* call, the checker would display a second message:

Missing: cursor(b) on b().

Finally, note how the implicit claims of the dependency assertion contrast with the weak claims of an invariance implied by the omission of an assertion. If we write $\alpha \leftarrow \alpha$, then the aspect expression α must be both read and written. It differs from omitting an assertion for α in two ways:

1. It claims that α may change, so if the procedure is called within an if-statement, a control dependence of α on the conditional is added.
2. It claims that α depends on the reference aspect expressions that determine the identity of the object associated with α.

As a result, the empty specification (claiming that all aspects are invariant) is satisfied by the empty implementation, *SKIP*. But no other Aspect specification passes *SKIP*, because an explicit Aspect assertion implies at least the writing of some object.

4.6 Summary

The chapter started with the notion of reference aspect dependencies. We saw why they are necessary and how they were implicit in the assertions we wrote in previous chapters. These extra dependencies strengthen the claims of a dependency assertion, and as a result, more bugs can be caught.

An Aspect assertion defines a set of possibilities. In marked contrast to conventional specifications, which say that some executions cannot happen, an Aspect specification says that some executions must happen. Aspect is thus good at detecting errors of omission. *SKIP*, the program that does nothing, satisfies only one Aspect specification: the empty one.

Now we have described the meanings of all three kinds of assertion:

- The dependency assertion $\alpha \leftarrow \beta$ means that, in some execution, the final value of α depends on the initial value of β, and, moreover, that β is read and α is written;
- The binding assertion $\alpha :- \beta$ means that each of the initial values of β is a final value of α in some execution;
- The allocation assertion x: t means that in some execution, the procedure allocates a fresh object of type t that is given the name x in the local scope of the specification; and

Aspect is a declarative language. The order in which the assertions are written is immaterial, and the meaning of an aspect expression in one assertion is not affected by any other assertion. We showed that bindings and dependencies in particular do not interact. Changes in the naming of objects cause no problem because all objects are named in the pre-state.

Chapter 5

Abstraction Functions

A procedure specification plays two roles in Aspect. For checking the procedure's implementation, it ac s as a criterion of correctness; for checking procedures that call it, it acts as a summary of expected behaviour.

A problem arises for abstract operations—the procedures of an abstract type, implemented in a cluster. These procedures are specified in terms of abstract aspects invented for the purpose of specifying the type. Yet they are coded in terms of a representation type, hidden from the outside, with its own set of aspects. How can we check an implementation against a specification written in a different vocabulary?

To bridge the gap, the programmer supplies an *abstraction function* that tells the checker how the abstract aspects are related to the representation aspects. The checker translates the criteria of the abstract specification into claims about the aspects of the representation. The translation is invisible to the user, who gives only the abstract specifications and the abstraction function. When an abstract assertion is violated, an error message is given in terms of the representation, where the deficiency lies.

The chapter starts by illustrating the idea of abstraction functions with a trivial example. We shall see that an abstract assertion implies an indeterminacy from the representation viewpoint that cannot be easily expressed with the existing notation, so a new assertion is introduced solely for the purpose of translation. After considering a variety of examples of abstraction functions, we shall see the general translation rule. The ideas are then applied to the buffer example of Chapter 2, and we see how some bugs can be detected in an implementation of a buffer operation specified there. Up until this point, the chapter deals only with dependency assertions. It ends with a short discussion of binding assertions.

Throughout the chapter, we are only concerned with the checking of an abstract operation's implementation. From the client's viewpoint, the abstraction function is irrelevant and only the abstract procedure specifications are needed.

5.1 Simple Examples

Consider an abstract type *poly* that models polynomials. Suppose we implement the polynomial as an array of coefficient/exponent pairs[1]:

rep = array [term]
term = record [coeff, exp: int]

Then we could represent the polynomial $x + 3x^2 + 5x^4$, for example, as

$$[1 : [1,1], 2 : [3,2], 3 : [5,4]]$$

The numbers preceding the colons are the indices of the array; this array is indexed from 1. Another valid representation of the same polynomial would be

$$[2 : [1,1], 3 : [5,4], 4 : [3,2]]$$

with the array indexed from 2 and the terms in a different order.

Consider an implementation of *poly* represented in this way (Figure 5.1). The implementations of two operations are shown: *get_degree*, which returns the largest exponent, and *add_term*, which adds a new term to the polynomial. The keyword *cvt* indicates an argument of the abstract type that will be viewed, inside the operation, in terms of its representation. So inside the operations, the type of *p* is not *poly* but *array[term]*.

Only part of the specification is shown. The abstract type has some aspects that include *degree*; *get_degree* obtains *degree(p)* and *add_term* changes it. The representation type, on the other hand, has no such aspect. Its aspects, from the specifications of arrays, records and integers are

[1]The keyword *rep* is used in CLU to name the representation type of a cluster; there is no need to say which type it represents because a cluster always implements exactly one abstract type.

```
poly = cluster is get_degree, add_term, ...
    %@ aspects degree, ...

    %@ abstraction
    %@     degree(A) ≪ R.el.exp
    %@     ...

    rep = array [term]
    term = record [coeff, exp: int]

    get_degree = proc (p: cvt) returns (int)
        %@ result ← degree (p)
        i: int
        if array[term]$empty (p) then return (0)
        else i := p[1].exp
            end
        for t: term in array[term]$elements (p) do
            if t.exp > i then i := t.exp
                end
            end
        return (i)
        end get_degree

    add_term = proc (p: cvt, coeff, exp: int)
        %@ degree(p) ← degree(p), coeff, exp
        %@ ...
        if coeff ~= 0 then
            rep$addh (p, term${coeff: coeff, exp: exp})
            end
        end add_term
```

Figure 5.1: Part of a polynomial implementation

size(p)— the size of the array,
low(p)— the low bound of the array,
el(p)— the elements of the array (a collection),
ind(p)— the order of the elements of the array,
exp(p.el)— the exponents of the terms (a pointer),
coeff(p.el)— the coefficients of the terms (a pointer),
p.el.exp— the value aspect of an exponent, and
p.el.coeff— the value aspect of a coefficient.

How then can we check the implementation of an operation like *get_degree*? We need to tell the checker what representation dependencies are implied by a dependency on *degree*. The abstraction function following the aspect declaration defines each abstract aspect by giving the representation aspects from which it can be derived. The *degree* aspect is defined as:

$$degree(A) \ll R.el.exp$$

which says that the *degree* of an abstract polynomial A is derivable from the value aspect of the representation object *R.el.exp*. The checker translates *get_degree*'s assertion

$$result \leftarrow degree(p)$$

into

$$result \leftarrow p.el.exp$$

which is satisfied by the implementation.

Checking the assertion of *add_term* is not so straightforward. If we just naively replace *degree(p)* by *p.el.exp* we will obtain

$$p.el.exp \leftarrow p.el.exp,\ coeff,\ exp$$

which is certainly not satisfied, since it implies[2] a mutation of the immutable integer object *p.el.exp*! To handle this case, we have to be more careful about the relationship between *degree(p)* and *p.el.exp*.

[2]Recall from Chapter 4 that $a(x) \leftarrow \ldots$ implies mutation of the object named x in the pre-state; the aspect a in this case is the anonymous value aspect of integers and the object x is the integer object *p.el.exp*.

5.2 The Nature of Abstract Objects

From the abstract viewpoint, a mutation of *degree(p)* has a simple meaning: the *degree* aspect of the object called *p* in the pre-state is modified. But from the representation viewpoint, there are two immediate problems. First, what object corresponds in the representation to *p*? Second, what corresponds to the *degree* aspect of that object?

1. The abstract object *p* does not correspond to any single object in the representation, but the whole structure of the array, its terms and the integer coefficients and exponents. We can imagine a shell drawn around these objects (Figure 5.2). From outside the cluster, the shell is opaque and a change to *p* might involve a change to any of these objects. A CLU procedure cannot change the binding of an argument to an object, so *p* will be bound to the same array object in the pre- and post-states. Moreover, a term added to the polynomial as a new record appended to the array will appear in the post-state to be part of *p*, even though it never existed in the pre-state. A dependency assertion therefore names representation objects in the ***post-state***. This is not reneging on the notion of prenaming; these objects become part of the abstract object named in the pre-state.

2. Now consider the notion of the ***degree*** aspect of ***p***. The abstraction function says that *degree(p)* is obtainable from ***p.el.exp***. That means that *degree(p)* may be changed by altering any aspect of the representation that affects the result of reading ***p.el.exp***. We could achieve this by adding a new term, changing the exponent of an existing term, or mutating the object ***p.el.exp*** itself (were it not immutable). The change can thus be at any level of the structure, since the abstract aspect is not associated with a single object of the representation.

There is a freedom here that is missing in standard procedure specifications. Usually, we have to specify exactly which object is mutated, since the calling context can determine (by aliasing) where the change is made to a structure. In the abstract case, however, there are no pointers[3] that cross the shell and the calling context cannot tell which object changes.

[3]This is an assumption; it is violated by *rep exposure*.

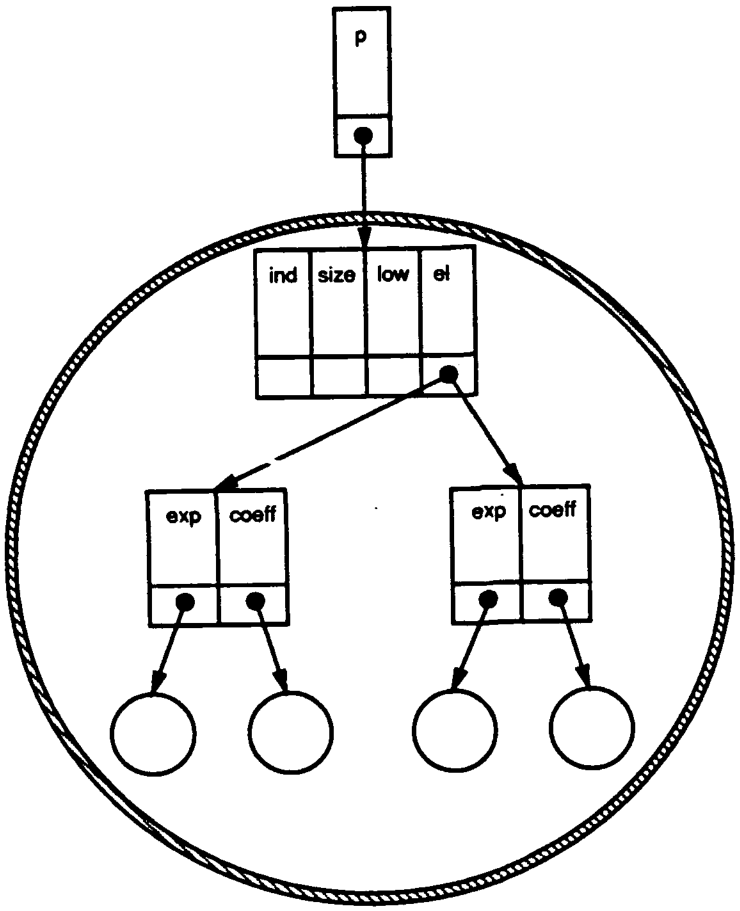


Figure 5.2: An abstract polynomial

It is therefore appropriate to introduce a new kind of dependency assertion, used only in the translation and not available for explicit use. Let

$$a(p) \leftharpoondown b(q)$$

mean that the *a* aspect of the object called *p* in the post-state, or any of its reference dependencies, depends on *b(q)*. Now we can translate the assertion of *add_term* into

$$p.el.exp \leftharpoondown p.el.exp,\ coeff,\ exp$$

The way it is satisfied by the implementation demonstrates both of the issues discussed above:

1. Prenaming of the abstract object implies post-naming of the representation object: the dependence on *exp* comes from an exponent object that is present only in the post-state and becomes part of the polynomial by the addition of a term.
2. An abstract aspect can be changed by modifying any of the representation objects on the path: the dependence on *coeff* is associated with the pointer aspect *el(p)* since *coeff* determines whether a term is added at all.

The dependence on *p.el.exp* simply arises from the existing exponent objects.

5.3 More Complex Abstraction Functions

If *get_degree* is called frequently, the representation we have chosen is inefficient. To improve the performance, we could replace it by:

rep = record[terms: array [term], deg: int]
term = record [coeff, exp: int]

in which the redundant *deg* field provides the degree of the polynomial without calling an array operation. The new abstraction function would have:

$$degree(A) \ll R.terms.el.exp \mid R.deg$$

This says that the degree is derivable from either the exponents or the *deg* field. Now the assertion *result* $\leftarrow$ *degree(p)* in the specification of *get_degree* is translated into

$(\mathit{result} \leftarrow \mathit{p.deg}) \vee (\mathit{result} \leftarrow \mathit{p.terms.el.exp})$

The disjunction operator $\vee$ is, like $\leftarrow$, part of the extended notation used only in translation. This compound assertion is satisfied if either disjunct is satisfied. The trivial implementation

return (p.deg)

satisfies the first, for example, and a clumsy version that still searches through the terms satisfies the second.

This new abstraction function will affect the checking of *add_term* in a different way. A requirement to change the dependences of *degree(p)* will now imply, in the representation, changes to both *p.terms.el.exp* and *p.deg*, ensuring that the redundancy in the representation is maintained:

$(\mathit{p.deg} \leftarrow \mathit{p.deg}) \vee (\mathit{p.deg} \leftarrow \mathit{p.terms.el.exp})$
$(\mathit{p.terms.el.exp} \leftarrow \mathit{p.deg}) \vee (\mathit{p.terms.el.exp} \leftarrow \mathit{p.terms.el.exp})$

The abstraction function thus incorporates some properties of what is usually called the *rep invariant*: a predicate on objects of the type that determines which of them correspond to valid abstract objects. The invariant here is that *p.deg* is equal to the largest *exp* field of the elements of *p.terms*. Although it cannot be expressed fully in Aspect, the fact that these are related is captured by the disjunction in the definition of *degree*.

Finally, suppose that we allow terms with zero coefficients, so that the polynomial $x + 3x^2 + 5x^4$ can be represented also as

$$[1 : [1,1], 2 : [3,2], 3 : [5,4], 4 : [0,9]]$$

The *degree* aspect is now derivable from the *combination* of the coefficients and the exponents:

$\mathit{degree}(A) \ll \mathit{R.terms.el.exp},\ \mathit{R.terms.el.coeff}$

In general, the assertion $\mathit{degree}(p) \leftarrow \ldots$ can now be satisfied with a change to either *p.terms.el.exp* or *p.terms.el.coeff*. As a final complication, if we allowed zero coefficients for the optimized representation we would have

$\mathit{degree}(A) \ll (\mathit{R.terms.el.exp},\ \mathit{R.terms.el.coeff}) \mid \mathit{R.deg}$

which says that *degree* is derivable either from the redundant degree field or from the combination of coefficients and exponents.

5.4 Rules for Translating Abstract Assertions

The translation rule is simple to generalize. Suppose we have abstract aspects a and b defined in the abstraction function as

$$a(A) \ll X_1 \mid X_2 \mid \ldots \mid X_n$$
$$b(A) \ll Y_1 \mid Y_2 \mid \ldots \mid Y_m$$

where the X_i and Y_i are lists of representation aspect expressions. If X_{ij} is the jth aspect expression in X_i, let $X_{ij}(p)$ denote the aspect expression obtained by replacing the R in X_{ij} by p. Also, let $B = \cup_i Y_i$.

Then the abstract dependency assertion

$$a(p) \leftarrow b(q)$$

is translated into

$$\forall i.\ \exists Y_{kl} \in B.\ \exists X_{ij} \in X_i.\ X_{ij}(p) \leftarrow Y_{kl}(q)$$

If either of a or b is not an abstract aspect of the type of the operation being checked, the rule degenerates in the expected way. If a is not abstract, it can be viewed as having the definition

$$a(A) \ll a(R)$$

which gives

$$\exists Y_{kl} \in B.\ a(p) \leftarrow Y_{kl}(q)$$

and, similarly, if b is not abstract, the translated assertion becomes

$$\forall i.\ \exists X_{ij} \in X_i.\ X_{ij}(p) \leftarrow b(q)$$

5.5 Buffers and Arrays

As a more realistic example of abstraction functions, we can apply these ideas to the buffer of Chapter 2. Since we shall represent the buffer with an array, we start by looking at the array specification (Figure 5.3).

Arrays have four aspects. Three are plain: *ind*, which represents the indexing of the array (the order of the elements), *size*, the number of elements

```
array = cluster [t: type] is new, addh, store, fetch, high, trim, ...
    %@ aspects
    %@     **el: t,          % refs to the set of elements
    %@     ind,              % the indexing of the elements
    %@     size,             % the number of elements
    %@     low               % the low bound

    new = proc () returns (array[t])
        %@ a: array[t]
        %@ result() :- a()

    addh = proc (a: array[t], e: t)
        %@ el(a) :- el(a), e()
        %@ ind(a) ← ind(a), low(a), size(a)
        %@ size(a) ← size(a)

    store = proc (a: array[t], e: t, i: int)
        %@ el(a) :- el(a), e
        %@ ind(a), el(a) ← ind(a), low(a), i

    fetch = proc (a: array[t], i: int) returns (t)
        %@ result() :- el(a)
        %@ result () ← ind(a), low(a), i

    high = proc (a: array[t]) returns (int)
        %@ result ← low(a), size(a)

    trim = proc (a: array[t], from, ct: int)
        %@ el(a) :- el(a)
        %@ ind(a), el(a) ← ind(a), low(a), from, ct
        %@ size(a) ← size(a), low(a), from, ct
        %@ low(a) ← from
```

Figure 5.3: A specification of arrays

```
buf = cluster is new, setCursor, ...

    %@ aspects text, clip, cursor, mark

    %@ abstraction
    %@     text(A) ≪ R.chars.el, ind(R.chars)
    %@     clip(A) ≪ R.cut.el, ind(R.cut)
    %@     cursor(A) ≪ R.csr | (R.csrX, R.csrY, R.chars.el, ind(R.chars))
    %@     mark(A) ≪ R.mk | (R.mkX, R.mkY, R.chars.el, ind(R.chars))

    rep = record [chars: ac,
                  csr, csrX, csrY: int,
                  mk, mkX, mkY: int,
                  cut: ac]
    ac = array [char]
```

Figure 5.4: Part of the annotated buffer code

and *low*, the low bound. The other aspect, *el*, is a collection representing a set of references to the element objects.

Only the operations used in the examples to follow are given: *new*, which returns a fresh array, *addh*, which appends an element at the high end, *high*, which returns the index of the top element, *store*, which replaces an element at a given index, *fetch*, which returns an element at an index, and *trim*, which resets the low bound to its *from* argument and removes the elements with indices below *from* or above $from + ct - 1$.

Figure 5.4 shows the part of the annotated *buf* cluster that contains the definition of its representation type. The text of the buffer (*chars*) is represented as an array of characters, and a similar array (*cut*) holds the contents of the clipboard. The cursor and mark are represented both as single integers and as coordinate pairs.

The abstraction function defines this relationship more precisely. The text of the buffer is derived from the characters in the *chars* array (*R.chars.el*) along with their order in the array (*ind(R.chars)*) . The clipboard is derived in the same way from the *cut* array. The cursor aspect, measured as the number of characters from the start of the buffer, can be derived directly from the value of the integer *csr* field. Alternatively, it can be derived from a combination of the *csrX* and *csrY* fields, along with the same aspects that gave the text (so that the effect of newline characters can be accounted for). The mark is derived similarly.

Figure 5.5 shows the annotated code for *cut_region*, an operation we specified in Chapter 2 that deletes the characters between the cursor and the mark of *b* and copies them into its clipboard. Arrays are assumed throughout to be indexed from 1. The array operations *store* and *fetch* appear in the code in short form:

> *txt[i]* = *c* stands for *ac$fetch(txt, i)* = *c*, and
> *txt[i]* := *e* stands for *ac$store (txt, i, e)*.

Rather than seeing how *cut_region* meets its specification, let us consider some bugs that could be caught. Most gross omissions would be detected. For example, forgetting to copy text to the clipboard, by omitting the *addh* operations in lines 20 and 29, would violate the assertion

> *clip(b)* ← *text(b)*

The checker does not display the abstract assertion in its error message. Instead, it displays

> *Missing: b.cut.el, ind(b.cut) on b.chars.el, ind(b.chars)*

giving a selection of representation dependencies as the cross product of two lists. A dependence of any aspect expression from the left list on any aspect expression from the right list will discharge the abstract assertion. Several small slips would have the same effect: omitting line 33, reversing *txt* and *cut* in the *addh* calls, etc.

Another gross omission would be to assume that the mark always precedes the cursor and leave out lines 8 to 13. This would violate the assertion

> *cursor(b)* ← *mark(b)*

and the checker would display

> *Missing: b.csr on b.mk, b.mkX, b.mkY, b.chars.el, ind(b.chars)*
> *Missing: b.csrX, b.csrY on b.mk, b.mkX, b.mkY, b.chars.el, ind(b.chars)*

```
cut_region = proc (b: cvt)
    %@ mark(b), cursor(b) ← mark(b), cursor(b)
    %@ clip(b), text(b) ← text(b), mark(b), cursor(b)
    m: int := b.mk
    c: int := b.csr
    if c = m then return
    elseif c < m then
        b.mk := c
        b.mkX := b.csrX
        b.mkY := b.csrY
    else
        b.csr := m
        b.csrX := b.mkX
        b.csrY := b.mkY
        m, c := c, m
        end
    txt: ac := b.chars
    cut: ac := ac$new ()
    disp: int := m - c
    last: int := ac$high (txt) - disp + 1
    i: int := c
    while i < last & i < c do
        ac$addh (cut, txt[i])
        txt[i] := txt[i + disp]
        i := i + 1
        end
    while i < last do
        txt[i] := txt[i + disp]
        i := i + 1
        end
    while i < c do
        ac$addh (cut, txt[i])
        i := i + 1
        end
    ac$trim (txt, 1, last - 1)
    b.cut := cut
    end cut_region
```

Figure 5.5: The implementation of a buffer operation

Two messages are produced because of the invariant implied by the disjunction in the definition of *cursor*. To correct the implementation, a dependency must be introduced for each of the two messages. If we set *b.csr* in line 9 but forgot to set *b.csrX* and *b.csrY*, we would get

Missing: b.csrX, b.csrY on b.mk, b.mkX, b.mkY, b.chars.el, ind(b.chars)

since changing *cursor(b)* involves changing more than one aspect of the representation.

A variety of bugs in quite complex code can thus be detected with a small procedure specification, because the abstraction function factors out the complexities of the representation. If the representation of *buf* is modified, only the abstraction function need be altered—all the operation specifications stay the same.

The abstraction function mitigates some other problems too. In its absence, we would have to write specifications in terms of the representation. These would be much longer than the abstract specifications. In the worst case, the number of assertions is the square of the number of aspect expressions. In this example, there are 24 aspect expressions that can be formed for a representation object, compared to 4 for the abstract object, so a concrete specification could be 36 times as long!

A concrete specification is also likely to be biased, making distinctions between implementations that are not observable. For example, we might assert that *b.csrY* depend on *b.mkY*. This precludes obtaining *b.csrY* by counting newline characters in *b.chars* as far as *b.mk*, which, despite its inefficiency would have the identical behaviour. The abstraction function provides the disjunction that is not available in procedure specifications.

The disadvantage of checking against the abstract specification is a loss of precision. Errors involving confusions of *b.csrX* and *b.csrY*, for example, could be detected if separate assertions were written instead of a single assertion for *cursor(b)*. Also, by specifying dependences of the sizes of the character arrays, we could catch errors like the omission of the *trim* call in line 32.

5.6 Abstract Binding Assertions

So far, we have only considered plain aspects. How are reference aspects of an abstract type treated? The stack of Figure 3.7, for example, has two

```
stack = cluster [t: type] is new, push, pop, top, size

    %@ aspects
    %@     size,              % number of elements in stack
    %@     *top: t,           % the top element
    %@     **rest: t          % the remaining elements

    %@ abstraction
    %@     size(A) ≪ size(R)
    %@     top(A) ≪ el(R)
    %@     rest(A) ≪ el(R)

    rep = array[t]

    push = proc (s: cvt, e: t)
        %@ top(s) :— e()
        %@ rest(s) :— rest(s), top(s)
        %@ size(s) ← size(s)
        array[t]$addh (s, e)
        end push
```

Figure 5.6: Stack representation

reference aspects: a pointer *top* and a collection *rest.*

Each abstract reference aspect must be mapped by the abstraction function to exactly one reference aspect of the representation. An example of an abstraction function for an array representation of stacks is shown in Figure 5.6. The binding assertions are translated directly into assertions over the representation aspects, so *push*'s specification would be translated into

```
el(s) :— e()
el(s) :— el(s)
```

which is discharged immediately by the *addh* specification.

The abstract specifications of the stack operations are unusual in being more precise than concrete specifications would be. The stack aspects distinguish the high element of the array from the others; this distinction can be maintained because the operations access the array in a disciplined fashion. This is an example of the specifier adding information that the checker could not have inferred itself, and as a result clients of the stack can be checked with a finer grain than if the array were used directly.

When there are abstract reference aspects, the abstraction function must obey a further constraint. The representation aspects they map to delineate the boundary of the abstract object: they point to objects outside. Any aspect expression that appears on the right-hand side of a definition in the abstraction function, on the other hand, contributes to an abstract property of the object. It is therefore essential that no aspect expression of an object that lies beyond the boundary of the abstract object be part of the definition of one of its aspects.

This phenomenon is called *rep exposure.* It can also arise because an abstract operation returns an object that is part of a representation object. This case is worse, since it cannot be ruled out by a static analysis of the abstraction function. However it arises, rep exposure is insidious because it provides a route by which an implementation can introduce dependencies that the checker cannot detect, leading to spurious bug reports.

5.7 Summary

The Aspect specification of a procedure has two purposes: the checking of the procedure itself, and the checking of its clients. An abstract operation is specified in terms of the aspects invented by the programmer for its type. These aspects are chosen to give the best possible checking of clients without over-complicating their specifications. But they cannot be used to check the implementation of the operation directly, whose dependencies range over a different set of aspects.

The programmer provides an abstraction function relating the abstract and representation aspects. Then all the operations can be checked by converting their specifications into claims about the representation. These claims include invariants that must be maintained by all the operations but which are specified only once. Changes to the representation force changes only to the abstraction function, and never to the specifications of the operations.

Chapter 6

Formal Semantics

This chapter gives an operational semantics for Aspect. It adds nothing that has not already been said informally, apart from some details left to intuition up until now. Almost all the tricky points of the semantics were explained in Chapter 4.

Since Aspect's view of a CLU program is abstract and unconventional, the aim of the semantics is not only to explain what a particular specification means, but to give the reader a firmer grasp of what kinds of thing can be specified at all. A second purpose is to lay a foundation for the checker, whose soundness rests on the relationship between its approximations and the semantics described here.

The semantics is given in three stages. First is the Aspect state, an abstract state over which Aspect specifications are defined. Second, the meaning of a specification is given as a relation on Aspect states. Third, the meaning of a CLU program is defined as an execution over Aspect states.

The specification semantics plays two roles. First, the state relation defines the possible behaviours of a procedure with no code, so that a program with calls to codeless procedures has a well-defined meaning. This absolves us from giving a semantics for the built-in types, because their operations are defined by specifications. Second, the state relation defines a notion of satisfaction. By calculating all the possible executions of a procedure's code, one can determine whether its specification is met. How to perform this calculation efficiently is discussed in Chapter 7.

6.1 The Representation of CLU Histories with Aspect States

An Aspect specification of a procedure is abstract; it does not distinguish the procedure's executions at the level of detail of a full conventional specification. Since the plain aspects have no values, Aspect regards as equivalent two states that differ only, for example, in the values of integer objects. It is convenient to define the meaning of Aspect specifications in an abstract domain in which these details are ignored.

The Aspect state is an abstraction of the CLU state. A single Aspect state corresponds to multiple CLU states: all those that retain the same basic structure but differ in the values of the plain aspects. Because Aspect is about dependencies, though, we shall need to consider sets of executions rather than individual states. In general, a pair of Aspect states corresponds to a set of CLU executions whose starting states (and ending states) share the same structure, and which have certain dependencies of the aspects of the ending state on the aspects of the starting state. To retain the notion of an Aspect state corresponding to a set of CLU states, we can regard the actual CLU states as "instrumented" with dependencies, so that a given state carries evidence of the history that preceded it.

Consider, for example, the statement

if b then a[i] := e end.

We might consider the evaluation or abstract execution of this statement from the Aspect state in which *a* has two elements, neither of which is the same as *e*. There are three Aspect states that could result: one with the first element of *a* replaced by *e*, one with the second replaced and one with neither replaced. The semantics will regard the choice between these outcomes as non-deterministic, since the values of *b* and *i* are not represented in the initial Aspect state. Now consider an evaluation from the Aspect state in which *e* is already one of the elements. There will still be three resulting Aspect states; two will share the same structure and will differ only in the dependencies of *el(a)*, the collection aspect of the array. The Aspect state thus distinguishes two CLU executions that end in the same state, because they have different dependencies.

$$\begin{aligned} \mathit{State} &= \mathit{Env} \times \mathit{Store} \\ \mathit{Env} &= \mathit{Var} \mapsto \mathit{Loc} \times \mathcal{P}\mathit{Source} \\ \mathit{Store} &= \mathit{Loc} \times \mathit{Aspect} \mapsto \mathit{Val} \times \mathcal{P}\mathit{Source} \\ \mathit{Val} &= \mathit{Unknown} + \mathit{Loc} + \mathcal{P}\mathit{Loc} \\ \mathit{Aspect} &= \mathit{PlainAspect} + \mathit{Pointer} + \mathit{Collection}. \end{aligned}$$

Figure 6.1: The domain equations of the Aspect state

6.2 The Domain Equations of the Aspect State

Like CLU, the Aspect state has an environment that binds locations to variables and a store that gives locations their values. The Aspect environment, however, is a simple mapping from variables to locations, and not a stack. This does not preclude the analysis of recursive procedures. A called procedure is viewed, in the code of its client, purely in terms of its specification which, being declarative, hides the internal details of the called procedure's code—including the stack frames it uses temporarily. The other distinctions are most easily seen in the domain equations (Figure 6.1). The symbol $\mathcal{P}$ means powerset, so $\mathcal{P}S$ denotes the set of all finite subsets of S.

A value is associated not with a location (which corresponds to an object in the heap) but with a particular aspect[1] of a location. There are three kinds of value

$$\mathit{Val} = \mathit{Unknown} + \mathit{Loc} + \mathcal{P}\mathit{Loc}$$

associated with three kinds of aspect

$$\mathit{Aspect} = \mathit{PlainAspect} + \mathit{Pointer} + \mathit{Collection}.$$

The plain aspects of a location (such as the size of an array or the colour of a window) have unknown values; the set *Unknown* contains the single element '?'. A pointer aspect represents a reference to a single object (like a field of a record) and a collection represents a reference to a set of objects (like

[1]The aspect names are assumed to be global. This is a simplification; in practice, types may share aspect names, so we can think of an aspect as tagged with the name of the type to which it belongs.

the elements of an array). Note that the value of a collection of a location is a *set* of locations, so Aspect cannot, for instance, discriminate the order of elements in an array. To distinguish the plain aspects from the reference aspects, we define

$$\mathit{Ref} = \mathit{Pointer} + \mathit{Collection}$$

The CLU environment maps variables to locations and not values. A CLU variable is a reference to an object, not a name for a pidgeon-hole (as in Pascal or Modula). Similarly, an object that contains other objects always refers to them indirectly. A record object, for example, will have some location with aspects corresponding to its fields. Each of these aspect/location pairs will have a value that is a location, and that location will hold the contained object.

The state is "instrumented" with dependencies. We shall see later, in the semantic definitions of assignment and the Aspect assertions, that the *sources* of the environment record, against each variable, what determined that it should point to that location. Similarly, the sources of the store record against each aspect of each location what determined its value. A source is essentially a marker that relates a component of the state to a component of an earlier state, and thus represents a dependency. It might seem odd to attach sources to individual states in this way, rather than describing dependencies in terms of state pairs. Nevertheless, it turns out to be a convenient and straightforward representation. The source type *Source* is not defined; it can be any set of labels large enough to distinguish the components of a state.

6.3 Aspect State Examples

To see how this formalizes the ideas we have discussed before, consider the *replace* procedure from Chapter 4, repeated in Figure 6.2. The procedure replaces the upper or lower buffer of the window *w* with the buffer *b* depending on the value of *cmd.*

The uppermost tables of Figure 6.3 show an Aspect pre-state for this procedure. It is like one of the graphs of Chapter 3 in tabular form. Note that the plain aspects have no values; only the structure of the state is constrained. The sources of the aspects are the initial tags described first

```
replace = proc (w: window, b: buf, cmd: string)
    if cmd = 'upper' then
        window$setUpper (w, b)
    elseif cmd = 'lower' then
        window$setLower (w, b)
        end
    end replace
```

Figure 6.2: A procedure to demonstrate Aspect states

in Chapter 2; they are numbered from 1 to 18 to distinguish them from the locations, which are numbered 100, 200, etc.

There are two possible Aspect post-states of *replace*. The lower table shows the Aspect state that results from the executions in which the first branch of the if-statement is taken. The value of *upper(w)* changes and it acquires dependencies on *b()*, *cmd()* and *cmd*.

6.4 Context

Only some Aspect states are well-formed at an arbitrary point in a program. These are determined by which variables are in scope, what their types are and which aspects are associated with those types. Whether a state is well-formed thus depends on its context. In this section, we look at how a context is built from the aspect declarations of the object types.

The context of a state is a tuple $\langle S, T, A \rangle$ where

- S is the set of variables in scope,
- T : *ObjectName* $\mapsto$ *Type* is a partial function from object names to types, an object name being a variable and a sequence of reference aspects

$$\mathit{ObjectName} = \mathit{Var} \times \mathit{Ref}^*$$

- A : *Type* $\rightarrow$ $\mathcal{P}$*Aspect* is a function that maps each type to the set of aspects associated with objects of that type.

Env		
Var	Loc	Sources
w	100	1
b	400	2
cmd	500	17

Store			
Loc	Aspect	Val	Sources
100	*upper*	200	3
100	*lower*	300	4
200	*text*	'?'	5
200	*cursor*	'?'	6
200	*mark*	'?'	7
200	*cut*	'?'	8
300	*text*	'?'	9
300	*cursor*	'?'	10
300	*mark*	'?'	11
300	*cut*	'?'	12
400	*text*	'?'	13
400	*cursor*	'?'	14
400	*mark*	'?'	15
400	*cut*	'?'	16
500	*value*	'?'	18

Env		
Var	Loc	Sources
w	100	1
b	400	2
cmd	500	17

Store			
Loc	Aspect	Val	Sources
100	*upper*	400	2,17,18
100	*lower*	300	4
200	*text*	'?'	5
200	*cursor*	'?'	6
200	*mark*	'?'	7
200	*cut*	'?'	8
300	*text*	'?'	9
300	*cursor*	'?'	10
300	*mark*	'?'	11
300	*cut*	'?'	12
400	*text*	'?'	13
400	*cursor*	'?'	14
400	*mark*	'?'	15
400	*cut*	'?'	16
500	*value*	'?'	18

Figure 6.3: A pair of Aspect states

The derivation of S is part of the CLU language definition and is not repeated here. We shall also assume that the types of the program variables are already known; our job is to add the typing of more general object names. Given the context in which the type of some variable v is t, that is

$$T[\![v]\!] = t,$$

the pointer aspect declaration

> *t = cluster is ...*
> *%@ aspects *e: rt, ...*

extends T by making rt the type of $v.e$:

$$T[\![v.e]\!] = rt.$$

Exactly the same rule holds for collection aspects. A set of clusters with aspect declarations thus defines a tree of object names for a given variable and a type for each legitimate object name. Finally, the aspects of a type are those listed in the aspect declaration of its cluster; from

> *t = cluster is ...*
> *%@ aspects* $a_1, \ldots, *e_1 : te_1, \ldots, **c_1 : tc_1, \ldots$

we obtain

$$A[\![t]\!] = \{a_1, \ldots, e_1, \ldots, c_1, \ldots\}.$$

6.5 Names of Locations

An object name defines a set of locations in a given state, obtained by viewing the name as a path through the store. For a variable, the set contains only the location delivered by the environment, but in general, because of collection aspects, an object name defines more than one location. The locations given by the object name u in state σ are given by:

$$locs[\![u]\!]_\sigma = \begin{cases} \{\sigma.env[\![x]\!].loc\} & \text{if } u = x,\ x \in Var \\ \bigcup_{l \in locs[\![v]\!]_\sigma} \{\sigma.sto(l,e).val\} & \text{if } u = v.e,\ e \in Pointer \\ \bigcup_{l \in locs[\![v]\!]_\sigma} \sigma.sto(l,e).val & \text{if } u = v.e,\ e \in Collection \end{cases}$$

6.6 Well-formed States

An Aspect state σ is well-formed at a particular point in the program if:

1. The environment maps exactly the set of variables S that are in scope:
$$dom(\sigma.env) = S$$
2. The value associated by the store with each location/aspect pair is of the right kind: unknown for a plain aspect, a location for a pointer aspect, a set of locations for a collection aspect.
3. It is well-typed. Let us say that a location l has a type t if it has an object name of type t, that is
$$\exists u.\ l \in locs[\![u]\!]_\sigma \ \&\ T[\![u]\!] = t$$
Then the state σ is well-typed if every reachable location has exactly one type.
4. Each location is divided into the aspects that are appropriate for its type. If l is a location with type t and a is an aspect, then
$$\langle l, a\rangle \in dom(\sigma.store) \ \Leftrightarrow\ a \in A[\![t]\!]$$

Since the type associated with a pointer or collection aspect may only be a cluster type, Aspect views the type of every location as a tuple of abstract types. There are no union types which would allow a reference aspect to point either to one type or another and so the state never has cycles. Of course a cluster may behave like a union type or be implemented by a union type, but its Aspect representation is always a tuple.

6.7 Procedure Specifications

It is convenient to define some meta variables for the syntactic categories given before:

$$\begin{aligned} a, b, c \in Aspect &= Plain + Ref \\ e, f, g \in Ref &= Pointer + Collection \\ x, y, z \in Var & \end{aligned}$$

An object name is a variable name and a sequence of reference aspects:

$$u, v, w \in \mathit{ObjName} = \mathit{Var} \times \mathit{Ref}^*$$

An aspect expression is either an aspect and an object name, or a variable name:

$$\alpha, \beta, \gamma \in \mathit{AspectExpr} = \mathit{Var} + (\mathit{Aspect} \times \mathit{ObjName})$$

The concrete syntax denotes aspect expressions as *x()* for a variable and *a(u)* or u (when the aspect is anonymous) for an aspect of an object. In addition to α, etc., we shall use *x()* and *a(u)* as meta expressions.

A procedure specification is a series of assertions, whose abstract syntax is:

$$\begin{aligned}
\mathit{ProcSpec} &= \mathit{Assertion}^* \\
\mathit{Assertion} &= \mathit{Alloc} + \mathit{Dep} + \mathit{Binding} + \mathit{Invar} \\
x{:}\,t \in \mathit{Alloc} &= \mathit{Var} \times \mathit{TypeExpr} \\
\alpha \leftarrow \beta \in \mathit{Dep} &= \mathit{AspectExpr} \times \mathit{AspectExpr} \\
\alpha :- \beta \in \mathit{Binding} &= \mathit{AspectExpr} \times \mathit{AspectExpr}
\end{aligned}$$

The semantics needs to be given only for elementary assertions of this form, because a compound assertion like

$$\alpha, \beta, \ldots \leftarrow \gamma, \delta, \ldots$$

is equivalent to the set of elementary assertions:

$$\alpha \leftarrow \gamma, \alpha \leftarrow \delta, \beta \leftarrow \gamma, \beta \leftarrow \delta, \ldots$$

Instead of defining the operator $\leftarrow$ ("affected by") to include the implicit dependencies of Chapter 4, we shall assume that each dependency, explicit and implicit, is given by a dependency assertion. The assertion $\alpha \leftarrow \beta$ in which $\leftarrow$ means "affected by" is thus replaced by

$$\alpha \leftarrow \beta,\ \alpha \leftarrow \gamma_1,\ \ldots,\ \alpha \leftarrow \gamma_i,\ \ldots,\ \alpha \leftarrow \gamma_n$$

where the γ_i are the reference aspect expressions of α and β and in which $\leftarrow$ now means "depends on". Recall from Chapter 4 that the reference aspect expressions of $a(x.e_1.e_2..e_n)$ are $e_n(x.e_1..e_{n-1})$, $e_{n-1}(x.e_1..e_{n-2})$, and so on, as far as $x()$.

We shall also assume that the implicit invariance of an omitted aspect α has been made explicit with the assertion $\alpha \leftarrow \alpha$, and that each binding assertion $\alpha{:}- \beta$ is accompanied by a dependency assertion $\alpha \leftarrow \beta$ when $\alpha \neq \beta$.

6.8 The Meaning of Aspect Expressions

In a dependency assertion, an aspect expression $a(u)$ denotes a set of sources. We can define the sources of the a aspect of the object in a different state from the state in which the object is named by u. So $sources[\![\alpha]\!]^{\sigma}_{\sigma'}$ is the set of sources of α in σ', with the naming of locations determined by σ:

$$sources[\![\alpha]\!]^{\sigma}_{\sigma'} = \begin{cases} \sigma'.env[\![x]\!].sources & \text{if } \alpha = x() \\ \bigcup_{l \in locs[\![u]\!]_{\sigma}} \sigma'.sto(l,a).sources & \text{if } \alpha = a(u) \end{cases}$$

In a binding assertion, an aspect expression denotes a set of values. Since binding is restricted to reference aspects, this value set is always a set of locations. So define $values[\![\alpha]\!]^{\sigma}_{\sigma'}$ to be the values of α in state σ', again under the location naming of σ:

$$values[\![\alpha]\!]^{\sigma}_{\sigma'} = \begin{cases} \{\sigma'.env[\![x]\!].loc\} & \text{if } \alpha = x() \\ \bigcup_{l \in locs[\![u]\!]_{\sigma}} \{\sigma'.sto(l,e).val\} & \text{if } \alpha = e(u), e \in Pointer \\ \bigcup_{l \in locs[\![u]\!]_{\sigma}} \sigma'.sto(l,e).val & \text{if } \alpha = e(u), e \in Collection \end{cases}$$

6.9 The Meaning of Assertions

Suppose now that we have a procedure with a signature

$$p = \ proc\,(arg_1{:}\,t_1, arg_2{:}\,t_2, \ldots)\ returns\ (t^1, t^2, \ldots)$$

annotated with a specification that is a sequence of assertions. In this section we shall look at binding and dependency assertions; allocations are treated later. The meaning of such a specification is a pair $\langle TR, MOD \rangle$ comprising a transition function that maps each pre-state to a set of post-states:

$$TR{:}\ State \rightarrow \mathcal{P}State$$

and a modification function that gives, for each pre-state, the set of location/aspect pairs that may be modified:

$$MOD{:}\ State \rightarrow \mathcal{P}(Loc \times Aspect)$$

The *MOD* function is used only to compute control dependencies.

The assertions define the transition function implicitly in the following way:

1. A dependency $\alpha \leftarrow \beta$ asserts that each source of β in the pre-state is a source of α in some resulting post-state:

$$\forall pre.\ sources[\![\beta]\!]_{pre}^{pre} \subseteq \left[\bigcup_{post \in TR(pre)} sources[\![\alpha]\!]_{pre}^{post}\right]$$

2. A binding $\alpha :- \beta$ has a similar meaning. It says that each location denoted by β in the pre-state must be included in those denoted by α in some resulting post-state:

$$\forall pre.\ values[\![\beta]\!]_{pre}^{pre} \subseteq \left[\bigcup_{post \in TR(pre)} values[\![\alpha]\!]_{pre}^{post}\right]$$

In addition to these constraints, there is a frame condition on TR. The arguments and their types define a set of locations in the store that is reachable by the procedure. No location outside this region may be affected at all—the stores of the pre- and post-states must match on these locations.

The dependency assertions define the modification function MOD. If the set of dependency assertions in the specification (before making the dependencies of invariant aspects explicit) is

$$\{a_i(u_i) \leftarrow b_i(v_i)\}$$

then the aspect/location pairs that might be modified from a pre-state of pre are

$$MOD(pre) = \bigcup_i (locs[\![u_i]\!]_{pre}^{pre} \times \{a_i\})$$

6.10 Discussion of Assertion Semantics

The above definitions are motivated by discussions in earlier chapters. Let us review some of the subtleties that arose, and see how they are reflected in the form of the semantics.

1. **Existence**. An Aspect assertion claims that some effect is possible in some execution of the procedure, not that it occurs in every execution. This is expressed in the semantics by forming the union over the post-states of the sources and values of α.

2. **Prenaming.** The naming of locations may differ in the pre- and post-states: after all, one of the reasons for calling a procedure is to change the structure of the store. To avoid complex interactions between assertions, all object names are interpreted in the pre-state, and *sources* must be defined over two states. The dependency assertion thus describes what happens to the sources of a given node, without concern for whether that node may undergo a change of name. In fact, it is essential to name objects in the pre-state because we might want to describe mutations of objects that have no names in the post-state (because they are no longer reachable from the procedure arguments or results).

3. **Invariance vs. Dependency.** The assertion $\alpha \leftarrow \alpha$ differs from the omission of an assertion for α in two ways. First, the dependency assertion implies additional pointer dependencies: since α is read and written, its final value is made dependent on the reference aspects of α. Second, it implies a possible change in the value of α (and thus contributes to *MOD*).

4. **Collections.** An aspect expression like *a(x.el)* may refer to several locations if *el* is a collection aspect. What then does an assertion like

   $$a(x.el) \leftarrow b(x.el)$$

   mean? The set union expression in the definition of the *sources* function resolves this by flattening the sets of sources. The assertion therefore says that any source of the *b* aspect of any location in *x.el* must be a source of the *a* aspect of some location in *x.el*.

   The union over post-states in the definitions causes a similar flattening. The assertion

   $$el(x) :- el(x)$$

   would otherwise imply that, in every execution of the operation, the objects in *el(x)* after include the objects in the *el(x)* before. The actual definition, in contrast, says that the set of post-objects in *el(x)* over *all* executions subsumes the sets before. In other words, any element that was in *el(x)* before will be in *el(x)* after in some Aspect execution of the procedure. Of course, there may be no CLU execution in which a particular element is in *el(x)* afterwards; the Aspect executions are a conservative estimate that generally includes more CLU executions than are actually possible.

This kind of assertion occurs in operations that delete some element from a collection, such as *stack[t]$pop* in Figure 3.7 or *array[t]$trim* in Figure 5.3.

6.11 The Meaning of Allocation Assertions

An allocation clause $a\!:t$ in a procedure specification causes an object to be allocated in the store in some location l, which is then assigned to the variable a in the environment.

The variable a is local to the specification. It has no significance in the calling context, since it is only used so that the allocated object can be linked to objects reachable from the arguments or to result objects. The semantics, therefore, must ensure that these local variables do not appear in the environment of the caller.

This is achieved by an existential quantification over these variables. Assume that a transition relation TR_0 has been defined for states in which the environment includes bindings for the allocated variables. Then define

$$hide(\sigma, A) = \langle \sigma.env \ominus A, \sigma.sto \rangle$$

which is the same state as σ but with the bindings of the allocated variables A removed from the environment. Finally, define the true transition relation to be TR, where

$$\begin{gathered} \sigma' \in TR(\sigma) \\ \Leftrightarrow \\ \exists \sigma_0, \sigma_0'.\ hide(\sigma_0, A) = \sigma \ \&\ hide(\sigma_0', A) = \sigma' \ \&\ \sigma' \in TR_0(\sigma_0) \end{gathered}$$

6.12 The Meaning of the Translation Dependency

The checking of abstract operations (Chapter 5) involved translating abstract dependency assertions into special assertions of the form $\alpha \leftharpoondown \beta$. To define the meaning of these assertions, we define $sources^*[\![\alpha]\!]_\sigma$ to be the sources of the aspect expression α and the sources of all the reference aspect expressions that determine the naming of the object given by α, in the state σ:

$$sources^*[\![\alpha]\!]_\sigma = \begin{cases} \sigma.env[\![x]\!].sources & \text{if } \alpha = x(),\ x \in Var \\ \bigcup_{l \in locs[\![v.r]\!]_\sigma} \sigma.sto(l,a).sources & \\ \quad \cup\ sources^*[\![r(v)]\!]_\sigma & \text{if } \alpha = a(v.r),\ r \in Ref \end{cases}$$

Then the meaning of $\alpha \leftharpoondown \beta$ is that the sources of β in the pre-state are included in the sources that determine the value of α in the post-state:

$$\forall pre.\ sources[\![\beta]\!]^{pre}_{pre} \subseteq \left[\bigcup_{post \in TR(pre)} sources^*[\![\alpha]\!]_{post} \right]$$

6.13 The Meaning of the Program Constructs

This section gives a semantics of the CLU program constructs in terms of Aspect states. In tandem with the semantics given above for specifications, this allows us to define the behaviour of code containing calls to procedures with specifications (and perhaps no code) and to judge a procedure's implementation against its specification.

The semantics is operational: it describes the possible sequences of states that can occur during the execution of a program. It uses Plotkin's scheme in which the meaning of a program is given by *evaluation relations* that are defined inductively over the syntax of the programming language. There are two evaluation relations, the state transition relation

$$\longrightarrow : Statement, State \mapsto State$$

and the modifies relation

$$\Delta : Statement, State \mapsto \mathcal{P}((Loc \times Aspect) \cup Var)$$

The assertion $\langle \sigma, S \rangle \longrightarrow \sigma'$, which says that the triple $\langle \sigma, S, \sigma' \rangle$ is in the transition relation, means that when the statement S is executed in state σ, there is an execution that terminates in the state σ'. Note that $\longrightarrow$ really is a relation and not a function. The execution is non-deterministic and there may be several σ''s for a given σ and S.

The assertion $\langle\sigma, S\rangle \;\Delta\; M$, which says that the triple $\langle\sigma, S, M\rangle$ is in the modifies relation, means that when the statement S is executed in state σ, the variables and location/aspect pairs of M may be modified.

Unfortunately, these relations cannot be defined independently. The modifies relation is needed to determine transition relation for an if-statement or a loop, because the control dependencies arise from the aspects that are modified. Also, aliasing make it impossible to calculate the modifies relation without the transition relation. The variables appearing syntactically in a statement do not limit the scope of the modifications; we need to know the object structure of the state. To make the interdependence of the relations clear, the rules are given side by side.

The relations are defined by inference rules. The rule

$$\frac{P, Q}{R}$$

says that given the premises P and Q, one can deduce the conclusion R. A rule without premises can be written without the line. The premises and conclusions are that certain tuples are in the relation. We are assuming that the relations contain exactly the tuples that can be inferred using the rules, and no more.

The simplest rules are for sequential composition:

$$\frac{\begin{array}{c}\langle\sigma, S_1\rangle \longrightarrow \sigma_1\\ \langle\sigma_1, S_2\rangle \longrightarrow \sigma_2\end{array}}{\langle\sigma, S_1; S_2\rangle \longrightarrow \sigma_2} \qquad \frac{\begin{array}{c}\langle\sigma, S_1\rangle \longrightarrow \sigma_1\\ \langle\sigma, S_1\rangle \;\Delta\; M_1\\ \langle\sigma_1, S_2\rangle \;\Delta\; M_2\end{array}}{\langle\sigma, S_1; S_2\rangle \;\Delta\; M_1 \cup M_2}$$

The rule for the transition relation (on the left) expresses the convention that execution proceeds downwards through the code statement by statement. S_1 and S_2 might be compound statements, so the effect of series of statements is obtained by applying this rule repeatedly. The modification rule (on the right) says that anything that might be modified when one statement is executed alone might be modified in the composition. Note the assertion on $\longrightarrow$ in the modification rule; the modifications of the second statement cannot be calculated without knowing what state it is executed in.

The if-statement rules are non-deterministic. Since the Aspect state hides the value of the conditional expression, we cannot tell which branch gets taken, so either may happen. There are two rules for each relation of the forms:

$$\frac{\begin{array}{c}\langle\sigma, S_i\rangle \longrightarrow \sigma_i \\ \langle\sigma, S_i\rangle \;\Delta\; M_i \\ \sigma' = cd(\sigma_i, \sigma, b, M_i)\end{array}}{\langle\sigma, \text{if } b \text{ then } S_1 \text{ else } S_2\rangle \longrightarrow \sigma'} \qquad \frac{\begin{array}{c}\langle\sigma, S_i\rangle \longrightarrow \sigma_i \\ \langle\sigma, S_i\rangle \;\Delta\; M_i\end{array}}{\langle\sigma, \text{if } b \text{ then } S_1 \text{ else } S_2\rangle \;\Delta\; M_i}$$

with $i = 1, 2$. The function cd adds the control dependencies. The state $cd(\sigma_i, \sigma, b, M_i)$ is obtained from the state σ_i by adding to the sources of the variables and the location/aspect pairs of M the sources of the variable[2] b in σ. This makes the changed components dependent on whatever b depended on in the state in which it was evaluated.

There are two while-loop rules for each relation. Again, we cannot determine the value of the condition, so it is always a possibility that the loop terminates without executing the body at all:

$$\langle\sigma, \text{while } b \text{ do } S\rangle \longrightarrow \sigma \qquad \langle\sigma, \text{while } b \text{ do } S\rangle \;\Delta\; \{\}$$

The other possibility is that the body of the loop is executed:

$$\frac{\begin{array}{c}\langle\sigma, S\rangle \longrightarrow \sigma' \\ \langle\sigma, S\rangle \;\Delta\; M \\ \langle cd(\sigma', \sigma, b, M), \text{while } b \text{ do } S\rangle \longrightarrow \sigma''\end{array}}{\langle\sigma, \text{while } b \text{ do } S\rangle \longrightarrow \sigma''} \qquad \frac{\begin{array}{c}\langle\sigma, S\rangle \longrightarrow \sigma' \\ \langle\sigma, S\rangle \;\Delta\; M \\ \langle\sigma', \text{while } b \text{ do } S\rangle \;\Delta\; M'\end{array}}{\langle\sigma, \text{while } b \text{ do } S\rangle \;\Delta\; M \cup M'}$$

It may help to read these backwards. The transition rule on the left can be read: "to execute *while b do S* in state σ, first execute S in state σ, then execute *while b do S* again from the resulting state, augmented with the control dependencies of M on b in σ'. Note that the control dependencies have to be added for each iteration, because the body of the loop can affect the aliasings. The execution of a statement can thus cause different modifications on different iterations.

The loop rules define the loop execution inductively. The first pair of rules, for termination, are the base case; the second pair give the recursion that unfolds the loop. By applying the rules for the recursive case for ever,

[2]This is a simplification: a conditional expression may be a procedure call with side-effects. Any if-statement can be translated easily into this form.

an infinite execution may always be obtained. This is another example of the Aspect semantics being a conservative estimate, generally allowing more executions than are possible. An Aspect specification is only concerned with the finite executions, so the infinite ones should be ignored.

Also, there is usually an infinite number of finite executions that end in infinitely many different states, because a loop body may allocate elements to a collection. Nevertheless, since there a finite number of variables and aspect, only a finite number of states can be distinguished by a specification. This is what makes Aspect tractable: the infinite set of possible states resulting from a procedure can be summarized by a finite set.

The transition rule for assignment shows how the environment is updated[3]:

$$\langle \sigma, x := y \rangle \longrightarrow \langle \sigma.env \oplus \{x \mapsto \sigma.env[\![y]\!]\}, \sigma.sto \rangle$$

The new environment is like the old one, but with x mapped to the location and sources of y. The result of the assignment is thus to make x and y aliases. CLU's assignment involves no copying of objects; in terms of Pascal, it is like an assignment on pointer variables. There is thus no change at all to the store. The modification rule makes this clear too; only the variable x is altered:

$$\langle \sigma, x := y \rangle \ \Delta \ \{x\}$$

Finally, the rules for procedure call, which use the specification of the procedure rather than its body:

$$\frac{\begin{array}{c} SPEC_p = \langle TR_p, MOD_p \rangle \\ \sigma_{pre} = \langle e_1, \sigma.sto \rangle \\ \sigma_{post} \in TR(\sigma_{pre}) \\ \sigma' = \langle e_2, \sigma_{post}.sto \rangle \end{array}}{\langle \sigma, x_1, x_2, \ldots := p(y_1, y_2, \ldots) \rangle \longrightarrow \sigma'} \qquad \frac{\begin{array}{c} SPEC_p = \langle TR_p, MOD_p \rangle \\ M = MOD_p(\sigma) \cup \{x_1, x_2, \ldots\} \end{array}}{\langle \sigma, x_1, x_2, \ldots := p(y_1, y_2, \ldots) \rangle \ \Delta \ M}$$

Let us assume that the formals of the procedure are called $arg_1, arg_2, \ldots, arg_n$. The transition rule breaks the call into three phases: the binding of the formals to the actuals, giving the environment

$$e_1 = \{\langle arg_i \mapsto \sigma.env[\![y_i]\!] \rangle \ | \ i \in 1..n\}$$

[3] $\oplus$ is functional override: $(f \oplus g)(x)$ is $g(x)$ when x is in the domain of g and $f(x)$ otherwise.

then the execution of the procedure itself (giving a state specified by *TR*); and finally a binding back of the result objects to the x_i, giving the final environment:

$$e_2 = \sigma_1.env \oplus \{\langle x_i \mapsto \sigma''.env[\![result_i]\!]\rangle \mid i \in 1..n\}$$

which is just the original environment altered by the assignment to the x_i.

The modification rule says that, in addition to the location/aspect pairs modified by the body of the procedure, the variables x_i are modified because of the assignment of the result objects. Recall that the *MOD* function maps states to location/aspect pairs only, indicating that the body of a procedure cannot modify the environment.

6.14 Specifications of Some Built-in Types

The reader may well feel cheated at this point. The semantics of the program constructs is almost independent of the structure of the Aspect state. Has something been swept under the rug? The answer is yes; the rug is the built-in types. A CLU program without procedure calls can do nothing. Assignment affects only the binding of variables to objects, and thus cannot affect any object; at the very least we need operations like + to construct new objects and operations like = to examine them.

Most of the complexity of the CLU semantics is thus given by the specifications of its built-in types. Here we shall look only at some representative examples. Simple immutable types have specifications like that of integers (Figure 6.4), based on a single anonymous aspect declared as "value".

It is not possible to specify CLU records because *record* is not a type: the number of fields varies, so it cannot be parameterized by a fixed number of type variables. If we take some syntactic liberties, however, a kind of specification can be written (Figure 6.5). We have already seen an array specification (Figure 5.3); together with integers and records, this defines a basic language[4].

[4]The checker, of course, has full specifications of all the built-in types, including strings, files, variants, etc.

```
int = cluster is add, times, equal, ...
    %@ aspects value

    add = proc (i, j: int) returns (int)
        %@ result ← i, j

    times = proc (i, j: int) returns (int)
        %@ result ← i, j

    equal = proc (i, j: int) returns (bool)
        %@ result ← i, j
```

Figure 6.4: Part of the built-in integer specification

```
record = cluster [t1, t2, ... : type] is set_c1, get_c1, set_c2, get_c2, ...
    %@ aspects *c1: t1, *c2: t2, ..., *ci: ti

    set_ci = proc (r: record, e: ti)
        %@ ci(r) :- e()

    get_ci = proc (r: record) returns (ti)
        %@ result() :- ci(r)
```

Figure 6.5: Part of the built-in record specification, simplified

6.15 Summary

We have seen in this chapter how the ideas of the preceding chapters can be formalized. We started by defining the Aspect state, an abstraction of a CLU computation. An object name composed of a variable and a sequence of reference aspects denotes a set of locations in a state; an aspect expression then denotes a set of values or a set of sources, according to whether it appears in a binding assertion or a dependency assertion.

Each assertion of a procedure specification defines a predicate over the abstract executions (pairs of Aspect states) of the procedure. The meaning of a series of assertions is just the conjunction of these predicates. By prenaming objects, we also ensured that the meaning of an object name in one assertion is unaffected by other assertions. The specifications are thus declarative and compositional.

Having defined the meaning of a procedure specification in abstract terms, we had to provide a similar view of CLU code so that the notion of satisfaction would be well-founded. The CLU semantics was far simpler than the specification semantics, because the complexity of CLU is embedded in the behaviour of the built-in types.

The purpose of the semantics is to clarify the meaning of an Aspect specification. There are two important applications of the semantics that are beyond the scope of the thesis. First, given a conventional formal semantics for CLU, we could prove a soundness theorem relating the two semantics, showing that an unsatisfied Aspect assertion is indeed always evidence of a bug. Second, we shall see in Chapter 7 that the checker approximates even the Aspect state, and thus has its own semantics. The correctness of the checking mechanism could be proved by relating that semantics to the semantics of this chapter.

Chapter 7

The Checker Mechanism

The semantics of Chapter 6 defines what it means for an implementation to meet a specification. Since the program constructs were defined operationally and the assertions were defined as predicates on states, checking might be just a question of performing all possible executions and then checking their final states against the assertions. Unfortunately, this is not possible. Because a loop body may allocate objects to a collection aspect, the Aspect state may grow without bounds, and in general there are infinitely many final states.

The first section of this chapter explains this problem in more detail, showing why some approximation is needed. The second section presents the particular approximation that has been implemented in the Aspect checker. There may be better approximations; this should be viewed only as demonstration that a simple one exists that leads to efficient checking without much loss of completeness.

The third and final section of the chapter describes some other features of the checker beyond the checking mechanism itself, such as storing and retrieving specifications, and gives an example of a session using the checker.

7.1 The Need for Approximation

Aliasing and dynamic allocation conspire to make unbounded states possible. Suppose we have a loop like

```
while b do
    e: t := t$new ()
    array[t]$addh (a, e)
    end
```

that allocates a fresh object in each iteration, extending an object by binding to a collection aspect. How many executions of the loop do we need to consider? The answer depends on how many names are available later in the code. If we had two variables, for example, that we could bind to different elements of the array, then we would have to consider states in which the array has three elements. Otherwise, having bound both variables, it would appear that mutating both of them would eliminate all existing dependencies of the elements, thus losing dependencies that could be associated with other elements.

This does not make the semantics intractable. There are, after all, only a finite number of possible object names in any given context, so there are thus only a finite number of distinguishable states. But it does rule out a simple minded execution of the semantics, since one cannot predict how many object names there will be later in the program.

7.2 The Implemented Approximation

Instead of performing *all possible executions*, the checker executes the procedure over a single approximating state that represents a (potentially infinite) set of Aspect states. By ensuring that the approximation is conservative—that is, it includes at least all the possible Aspect states—the checker can guarantee that any required state missing from the set given by the final approximating state must also be missing from any actual final state. Some bugs will go undetected because of the approximation, but the error report will still be free of spurious listings.

7.2.1 The Approximating State

The approximating state represents a set of Aspect states in two ways. First, a pointer aspect may have several values that correspond to possible rather than certain references. So two states in which a is aliased to b in one and c in the other are represented as a single state in which a points to both b and c. Second, a set of objects referenced by a collection aspect may be merged into a single object. The first approximation is necessary to represent a set of Aspect states as a single state; the second, and more critical, approximation is needed to represent an infinite set.

If two objects are pointed to by the same set of pointer aspects, they cannot be distinguished. They may then be merged together: one is discarded and the dependencies and values of its aspects are added to the other's. This "garbage collection" must make sure not to discard an object that may be aliased in the calling context. The approximating state therefore marks those objects that existed in the pre-state of the procedure and never discards them. This marking is information that is not present in any of the Aspect states that an approximating state represents, although it could be obtained from the pre-state.

Figure 7.1 shows the definition of the Aspect state from Chapter 6 and, below it, the definition of the approximating state used by the checker. The additional component *Multilocs* marks the objects that represent collections and *Prelocs* marks the objects that existed in the pre-state. Note the value of a reference aspect is always a set of locations in the approximating state and the environment maps variables to sets of locations too. There is still, however, only one set of sources for the variable; they represent the dependencies that determine which location the variable names. The store likewise has only a single set of sources for each location/aspect pair.

Each approximating state Σ represents some set of Aspect states $\gamma(\Sigma)$. One can think of obtaining these in two stages. First, each collection object in *Multilocs* is expanded into a set of objects whose dependency and value sets are subsets of those of the collection object. Each possible expansion gives a different state. Second, each of these expanded states is trimmed by removing all but one of the values of each pointer aspect, and taking subsets of the dependencies. Again, each possible trimming gives a different state.

Information is lost in the approximation because $\gamma(\Sigma)$ includes more Aspect states than it should. If we took two Aspect states σ_1 and σ_2, and approximated them with the state Σ_{12}, we would find many other states besides them in $\gamma(\Sigma_{12})$.

For dependencies, this loss of information is inconsequential, since the dependencies are flattened anyway in the Aspect specification against which the final state will be checked. It is not possible to say that an aspect has either some set of dependencies or some other set; the dependency assertion gives a lower bound on the total set of dependencies.

In contrast, the approximation is damaging for the value sets. Suppose that in σ_1 the variable x names object O and y names P, and in σ_2, x names object P and y names O. Then Σ_{12} will assign both O and P to x and likewise

The Aspect State:

$$\begin{aligned}
State &= Env \times Store \\
Env &= Var \mapsto Loc \times \mathcal{P}Source \\
Store &= Loc \times Aspect \mapsto Val \times \mathcal{P}Source \\
Val &= Unknown + Loc + \mathcal{P}Loc \\
Aspect &= PlainAspect + Pointer + Collection
\end{aligned}$$

The Approximating State

$$\begin{aligned}
State &= Env \times Store \times Multilocs \times Prelocs \\
Env &= Var \mapsto \mathcal{P}Loc \times \mathcal{P}Source \\
Store &= Loc \times Aspect \mapsto Val \times \mathcal{P}Source \\
Val &= Unknown + \mathcal{P}Loc \\
Aspect &= PlainAspect + Pointer + Collection \\
Multilocs &= \mathcal{P}Loc \\
Prelocs &= \mathcal{P}Loc
\end{aligned}$$

Figure 7.1: Approximation of the Aspect State

to y, including a state in which both x and y name O. A possible aliasing of x and y is thus implied even though it cannot occur in any execution. Bogus dependencies might be inferred as a result and a bug missed.

7.2.2 Executing Code

Recall, from Chapter 6, that a procedure specification claims that some set of post-states are possible outcomes of *every* pre-state. A complete check would require evaluating the code from all possible pre-states. The checker evaluation uses only a single state, so an approximating state must be chosen that is likely to catch the most bugs. This turns out to be the state with as little uncertainty as possible: each variable and pointer aspect has a single value and the are no internal aliasings of the arguments.

All the state changes that occur in the execution are due to procedure calls or assignment statements. Procedure calls are handled by executing their specifications, described in the next section. Assignments are executed according to the semantics of Chapter 6, except that a variable's entry in the environment acquires a set of values rather than a single value.

An if statement can lead to two states, one for each branch. To include both of these in one approximating state, the branches are executed separately and the resulting states are *merged* together. The merge of two approximating states Σ_1 and Σ_2 must represent at least the union of the Aspect states they represent:

$$\gamma(merge(\Sigma_1, \Sigma_2)) \supseteq \gamma(\Sigma_1) \cup \gamma(\Sigma_2)$$

The simplest way to perform the merge is to amalgamate the components of the two states. Since they are in the scope, they must have the same variable sets. The new environment is obtained by giving each variable the union of the values and sources of the two states. The new store, similarly, is formed by unioning the values and sources of the common aspect/locations, and adding the mappings for those locations that are allocated in only one of the two states.

Two problems arise in handling loops: finding a fixed point and preventing unbounded growth of the state because of allocations within the loop body. To find a fixed point state that accumulate the effects of any number of iterations, the checker merges the states resulting from zero, one, two iterations, etc. This process stops when the accumulated state no longer changes.

Two states may be equivalent even if they are not the same. If there are two allocated locations that are not in *Multilocs* (that is, do not represent collections of objects) with the same parents (that is, pointed to by the same location/aspect pairs or variables), they cannot be subsequently distinguished. The checker therefore does *compaction*, reducing the state to a canonical form in which allocated locations are merged, by removing one and adding its values and sources to the other. Also, locations may become unreachable; these are just lopped off.

Compaction is always performed after each merging of two states. Since the accumulated loop state grows monotonically, the fixed point can be found cheaply by a simple trick. The checker keeps a count of the number of values and sources in the state and stops executing the loop when it no longer changes.

Compaction solves the problem of unbounded growth too, since the offending allocated locations are merged into existing locations.

7.2.3 Executing Specifications

To execute a procedure call, the checker looks up its specification and executes that instead. In Chapter 6, we gave the meaning of an assertion as a claim that the procedure results in *at least* certain post-states. To catch as many bugs as possible, we assume the weakest valid implementation of the called procedure, and execute the specification to give exactly the minimal set of post-states.

There are three complications. First, since the object names that appear in aspect expressions all refer to locations in the pre-state, and since executing an assertion may change the structure of the state, the checker must start by constructing a mapping of object names to locations, and only then execute the specification.

Second, the assertions are conjunctive; if one says that an aspect should have the dependency sources S_1 and another says the same aspect should have S_2, the checker must satisfy both assertions and give it $S_1 \cup S_2$.

The third complication is a result of the approximation. Suppose we want to execute the dependency assertion

$$a(x) \leftarrow b(y)$$

in an approximating state in which the variable x points to two locations. In one of the states represented by the approximating state, x does not name one of those locations, so we cannot just replace its sources with the sources of $b(q)$. Because of the uncertainty, the checker ***adds*** the sources of $b(q)$ to the sources of the a aspect of each location.

The sources of an aspect expression must be obtained from all the locations it may refer to. For example, the set for $b(q)$ is the union of the source sets of the b aspect of all the locations called q.

As a general rule, values and sources are added to and not replaced whenever the object name on the left-hand side of the assertion refers to more than one location. Thus, if the location is in the set *Multilocs* it represents a collection of locations, and the same conservative approach is followed.

7.2.4 Checking The Final State

The result of executing the procedure is a single final approximating state. Each of the assertions in the specification is checked against this state, and any omitted values or sources are displayed as bug messages.

Allocation assertions cannot be checked directly (especially since the checker can discard allocated locations from the approximating state if their presence is not observable). Instead, the dependencies and values of their aspects are checked indirectly. Suppose, for example, that we have a specification that contains these assertions:

$$n\colon t$$
$$a(n) \leftarrow b(x)$$
$$r(y) :- n()$$

Although we cannot find the allocated location n, we can infer (from the binding) that either it is called $y.r$ in the post-state or it has been absorbed into $y.r$. The dependency assertion thus implies that the a aspect of the location called $y.r$ (in the post-state) has a dependency on $b(x)$, which is easily tested.

7.3 Using the Checker

The checker has several functions: storing and retrieving specifications, checking specifications for consistency and, of course, checking code against specifications. It is implemented in CLU in about 15,000 lines of code.

The checker runs in three phases. First, the annotated code is parsed and the specifications are analyzed for simple consistency properties (principally that aspect expressions in procedure specifications are well-formed). The specifications are saved as part of the state of the checker. Second, the usual compiler static analysis—including type checking—is performed. Third, the code of each procedure is checked in turn against its Aspect specification, using the saved specifications for procedures it calls.

Aspect specifications may be organized in libraries. To set up a library, the user runs the specification-reading phase of the checker on some CLU programs. The checker extracts type and procedure specifications from the code, parses and checks them and saves them in its internal state. The user then requests that the specifications be dumped to a library file in a compacted form. When, in a later invocation of the checker, the user wants to check a program that uses these specifications, the library file is loaded and the specifications become immediately available. The built-in types and operations are pre-specified and are automatically loaded when the checker is invoked.

A sample interaction with the checker is shown in Figure 7.2. I started by invoking the checker in the Unix shell. Then, at the "command:" prompt, I told the checker to load the specifications of the integer set in the file called "intset.spc". I then invoked the checking of the *remove_dupls* procedure[1] in the file "remove_dupls". The checker examines each kind of termination separately. Since the procedure has no exceptions, there is only the normal flow of control to check. The assertion on line 3 is found to be violated.

7.4 Summary

The Aspect semantics is not tractable because the combination of aliasing and dynamic allocation lead to unbounded states. Checking can still be performed with little loss of completeness by representing the set of possible Aspect states with a single approximating state. This is not by any means the only way to approximate the Aspect semantics. I have never found the loss of information to be significant, but there may be applications in which more precision is required. Perhaps, on the other hand, a cruder approxima-

[1]Discussed in Chapter 1.

```
unix> aspect

***** Aspect Checker Version 3, 1/1992 *****

command: spec intset
command: check examples

   Making Aspect specs for /u/jackson/research/aspect/exe/intset.spc

   Reading spec of view for cluster intset
   Reading spec of procedure/iterator create
   Reading spec of procedure/iterator insert
   Reading spec of procedure/iterator member

   time = 0.300 seconds

   External     Referencing Modules
   ---------------------------------------------------------
   intset       intset

command: ch rem-dupls

   Checking /u/jackson/research/aspect/exe/rem-dupls.clu

   Reading spec of procedure/iterator remove_dupls

   Checking flows for remove_dupls
   ... normal flow...
   3      BUG    low(a) <- low(a) not found.

   time = 1.630 seconds

   External        Referencing Modules
   ---------------------------------------------------------
   intset          remove_dupls
   remove_dupls    remove_dupls

command: quit
unix>
```

Figure 7.2: A sample checker run

tion would do instead, allowing a simpler implementation and improving the checker's performance further.

Chapter 8

Extensions

This chapter starts by relaxing two constraints that we have assumed so far. The first is that the aspects of a type be independent; the second—a more drastic one—that the user write specifications of called procedures at all. It then discusses some limitations of Aspect and speculates on how they may be overcome.

8.1 Aspect Orderings

The bug detection scheme assumes that the aspects of a type are independent: that one cannot be obtained from any combination of the others. What goes wrong if the aspects are not independent? Suppose the buffer type includes an aspect *size* representing the number of characters in the buffer, and that we want to check the procedure *IsEmpty* (Figure 8.1) that returns true when the buffer contains no characters. It works by calling the buffer operation *getText* that returns the text of the buffer as a string.

Without any information about the relationship between *text(b)* and *size(b)*, the checker will infer that the assertion of *isEmpty* is not satisfied, because *result* depends on *text(b)* and not on *size(b)*.

To remedy this problem, the checker must be told that the size of a buffer is derivable from its text. This information is provided as an *order* annotation that follows the aspect declaration for the cluster:

> *aspects text, clip, mark, cursor, size*
> *order size* $<$ *text*

```
isEmpty = proc (b: buf) returns (bool)
    %@ result ← size(b)
    s: string := buf$getText (b)
    return (string$empty (s))
    end isEmpty

getText = proc (b: buf) returns (string)
    %@ result ← text(b)
```

Figure 8.1: The problem of dependent aspects

The checker then allows a dependence on *text(b)* as a substitute for a dependence on *size(b)* and no spurious reports are generated.

Adding extra dependent aspects is useful because it allows more bugs to be caught. With *size* included in the aspects of *buf*, we can distinguish operations that only provide the *size* of a buffer from those that provide the *text* too. Figure 8.2 shows two implementations of a procedure that is intended to count the number of characters between the cursor and the end of the line. One is faulty; it calls the operation *num_following* that gives the number of characters until the end of the whole buffer. Without the *size* aspect, it could not be ruled out, because there would be no way to specify that *num_following* gives less information about the buffer than *search*.

Adding derivable aspects need change only one part of the checking mechanism. When the checker compares the code dependencies to the dependencies of the specification, it can apply substitution rules to find out whether a required dependency really is missing.

There is a simpler way to do this, however, which requires no change to the checker mechanism at all—only to the initial state of the procedure being checked. Because the relationship between aspects is always a partial order, we can represent it in the tagging of the aspects. In the initial state of the buffer object (Figure 8.3), the *text* aspect has two tags, one of which is the same tag as *size*. Making an aspect depend on *text(b)* will now cause it to acquire both tags, so that it automatically bears a dependency on *size(b)*. Each tag can be thought of as a lump of information; *text(b)*'s information subsumes *size(b)*'s, so its tags do too.

```
num_to_eol_good = proc (b: buf) returns (int)
    %@ result ← cursor(b), text(b)
    i: int := buf$search (b, newline_char)
    return (buf$getCursor (b) − i)
    end num_to_eol_good

num_to_eol_bad = proc (b: buf) returns (int)
    %@ result ← cursor(b), text(b)
    return (buf$num_following (b))
    end num_to_eol_bad

num_following = proc (b: buf) returns (int)
    %@ result ← cursor(b), size(b)
```

Figure 8.2: Catching more bugs with dependent aspects

text	clip	cursor	mark	size
① ⑤	②	③	④	⑤

Figure 8.3: The initial tagging of the buffer aspects

There may be several aspects derivable from a single aspect, and further aspects derivable from these. An aspect must then carry the tags not only for the aspects directly derivable from it, but also for the aspects derivable from them, and so on.

The aspect ordering seems indispensable. It is used in the specifications of the built-in types; arrays, for example, include an emptiness aspect (omitted in the simplified specification of Figure 5.3). Nevertheless, the ordering notion is not as flexible as it should be. It would be helpful if arbitrary relationships could be introduced, allowing aspects to be derived not only from other individual aspects but also from combinations. The array specification, for instance, might benefit from having both *high* and *low* aspects, but this is forbidden. We cannot express the notion that *size* is derivable from *high* and *low* in combination; the partial order only allows an aspect to be derivable wholly from another aspect, or from any one of several.

8.2 Omitted Specifications

A procedure can be checked even if the specifications of procedures it calls are missing. The checker must assume that each missing procedure has every possible effect: causing dependencies, establishing aliases and allocating objects.

This approximate specification is constructed from the procedure's header as follows. A set of names for the objects reachable by the procedure is obtained from the argument and result variables and the aspect declarations of their types. Each of these object names has an associated set of aspects, and thus a set of aspect expressions, E say. E includes also the appropriate pointer aspects of the environment itself: $arg_i()$ for each argument arg_i and $result_i()$ for the ith result.

The aspect expressions are then partitioned according to their type. The plain aspect expressions are placed in E_0, say, and the reference aspect expressions in E_1, E_2, etc., according to their types t_1, t_2, etc. Lastly, the checker adds variable names n_{i1}, n_{i2}, etc., for each type t_i, naming a set of allocated objects, one for each argument object of each type, and includes their aspect expressions in E. The specification then contains:

- An allocation assertion $n_{ij} : t_i$ for each allocation variable of type t_i.

- A dependency assertion $e_1 \leftarrow e_2$ for each pair of aspect expressions $e_1, e_2 \in E$, where e_2 is not an expression of a result or allocated variable and e_1 is not an argument aspect of the environment.

- A binding assertion $e_1 :- e_2$ for each pair of aspect expressions $e_1, e_2 \in E_j, j > 0$ with the same type, again with the restriction that e_2 is not an expression of a result or allocated variable and e_1 is not an argument aspect of the environment.

Sometimes a specification generated in this manner works fine; sometimes the loss of information prevents bugs from being detected. Omitting a specification is least damaging when the procedure has few arguments, each of which has little structure, and when the procedure's role is disjoint from the other procedures. In particular, it makes sense to omit the specification of an entire type.

The *remove_dupls* procedure of Section 1.2 is an example where omission is successful. None of the set operations need be specified, since they play no role in the critical aspect dataflow. The set cluster need also have no aspect declaration. When specifications are not provided for the operations of a cluster, an aspect declaration is worthless and the checker assigns the type a single anonymous aspect.

Another early example illustrates the opposite. In Section 2.6, to find a bug in *exchange*, we needed a specification of the called procedure *setMarkAtCursor*. Even if its code were inlined, the bug could not be found, because the specification expressed information that could not be tractably inferred from the code.

Generating specifications from procedure headers alone may not be adequate. In many cases, a precise specification is unnecessary, but a specification that allows all effects will introduce so many dependencies and aliases that checking will be confounded. There are some simple ways in which much better specifications can be generated. If the Aspect specification of a type includes whether or not its objects are mutable, the checker can omit assertions that claim to change immutable objects. Otherwise the generated specification for *set$member(s, e)*, for example, allows modification of the integer *e*. In the *remove_dupls* example, this happens to cause no problems but it might be troublesome elsewhere.

```
stack = cluster [t: type] is new, push, pop, top, size

    %@ aspects size, *top, **rest

    push = proc (s: stack[t], e: t)
        %@ top(s) :– e()
        %@ rest(s) :– rest(s), top(s)
        %@ size(s) ← size(s)

    pop = proc (s: stack[t]) returns (t)
        %@ result() :– top(s)
        %@ top(s), rest(s) :– rest(s)
        %@ size(s) ← size(s)
    ...
```

Figure 8.4: Specification of a stack

8.3 Polymorphism Problems

We saw an example of an Aspect specification of a polymorphic type in Section 3.7: a stack whose elements could be of any type. Part of the specification is shown again in Figure 8.4. This kind of polymorphism needs no special treatment. When the stack is instantiated, the type variable is bound to the element type; the checker constructs a stack object from this specification and the specification of the element type. No special interpretation of the operations is necessary, since they do not even mention objects of the unknown type. The aspect expression *e()* appearing in *push* is not, remember, an aspect of the variable e, but a pointer aspect of the environment. The specification of *push* talks about the identity of the object e, but no properties of the object itself.

This works fine for containers like stacks, lists, trees, queues, etc., in which there are no interactions between the containing object and the elements themselves: they are just inserted and removed. But for other kinds of polymorphic objects, properties of the elements might affect the behaviour of the container.

```
queue = cluster [t: type] is new, enq, ...
    where t has value: proctype (t) returns (int)
    %@ aspects **el: t, ord, ...

        rep = array [t]

    new = proc () returns (cvt)
        return (rep$new())
        end new

    enq = proc (q: queue[t], e: t)
        %@ ord(q) ← ord(q), ??
        if value(e) > value(q[1]) then ...
        ...
        end enq
```

Figure 8.5: Part of a polymorphic queue implementation

A priority queue, for example, might express the order of the elements with an aspect *ord* (Figure 8.5). The *enq* operation will determine the position of the new element in the queue according to some valuation of that element by calling an operation *t$value* of the element type.

The behaviour of *enq* cannot be fully known until the type is instantiated and *t$value* names an actual procedure. By convention, the *t$value* operation has certain properties (like defining a total order on the element type) but it may not: it could even mutate its arguments.

Type checking the code of *enq* is tricky since it calls an unknown procedure. CLU addresses this problem by allowing the signature of such an operation to be declared in a *where* clause following the cluster header. The type checker can then work in a modular fashion. In checking the *queue* cluster, it assumes that *t$value* has the signature given in the *where* clause. When *queue[t]* is instantiated by binding the type variable *t* to some real type T, it checks that T has the operation *value* and that it has the right signature.

The *enq* operation cannot be specified in Aspect. What would the dependency assertion say for *ord(q)*? The current version of Aspect would only allow something like

ord(q) ← - *ord(q), e()*

which says that the ordering after depends on the object identity of the object inserted. But we want to say that it depends on some property of the object *e* itself. This cannot be done, because we do not even know what the aspects of *t* are. It is not clear how to solve this problem, but it will probably involve attaching Aspect specifications of some kind to the *where* clause.

8.4 Over-Specification Due to Immutable Objects

Aspect has no special treatment for immutable objects. This can lead to over-specification: an Aspect assertion may make finer distinctions than are observable in CLU. For example, if *i* is a positive integer, the binding

result() :— i()

is satisfied by *return (i)*, but not by

```
j: int := 0
while true do
    if i = j then return (j) else j := j + 1 end
    end
```

because the object *j* (according to Aspect) is never the same object as *i*. But the two cases are indistinguishable: if two integers are equal, there may be no way to tell whether they are the same object.

Binding assertions cannot be eliminated because there is no other way to express the equality of the aspects of different objects. Also, without binding assertions, specifying that one object acquires each of its aspects from another object demands a dependency assertion for each aspect.

Perhaps a binding assertion should have a different meaning for immutable objects. It should imply only that the bound objects have equal value. Mention of the identity of an immutable object should be forbidden, or it should somehow be converted into an aspect denoting its value. This means, at the very least, that the immutability of a type has to be specified. This information would be useful anyway for generating specifications when they are missing (Section 8.2).

```
pair = record [one, two: foo]

funny_set = proc (p, q: pair, f: foo)
        %@ one(p) :- f()
        %@ two(q) :- f()
        pair$set_one (p, f)
        pair$set_two (q, f)
        funny_set
```

Figure 8.6: An example of over-specification

8.5 Over-specification Due to Aliasing

Binding assertions are sometimes too strong, and there is no way to express a weaker specification that still makes sense. Recall (from Section 6.8) that a binding assertion gives the values of the post-state aspect for *every* possible pre-state. Suppose we specify a procedure that binds an object to the first field of one pair and the second field of another (Figure 8.6). The first binding assertion says that, whatever the pre-state, the value of the *one* field of p in the post-state is equal to the identity of the object f. The omission of bindings for *two(p)* and *one(q)* implies their invariance.

This specification is not satisfied by the given implementation, because if p and q are aliased, a change to *one(p)* is a change to *one(q)* also, so the implicit invariance of *one(q)* is violated.

To resolve this problem, Aspect could allow different bindings to be specified for different aliasings of the pre-state, or a precondition saying that the procedure's behaviour is unconstrained when certain aliases are present. This situation does not seem to arise much in practice, however. Moreover, the checker would not give a spurious message anyway, because it only examines the procedure for initial states without internal aliasing.

8.6 Values for Plain Aspects

Aliasing forced us to give values to reference aspects (Chapter 3). We could equally well have given values to plain aspects; the distinction was a pragmatic one. In fact, there are several cases in which it would be useful to do this.

An *insert* operation on a container object that has an aspect *emp* (whether the container is empty or not) might currently have the dependency assertion

$$emp(a) \leftarrow \emptyset$$

which says, effectively, that the emptiness of the array is set to constant. If the plain aspect *emp* were to have values *true* and *false*, we could say that the array is non-empty after:

$$emp(a) :\!- \mathit{false}$$

So long as there are a finite number of values of a plain aspect, this should be no harder to check than regular bindings. But the checker could now treat the conditional expressions of if-statements and loops differently, branching appropriately when the expression has only a single value.

Giving values to plain aspects becomes much more useful if conditional assertions may be written. Suppose we specify a queue with two aspects: *front*, a pointer aspect to the element at the front of the queue and *rest*, a collection of the remaining elements. We might naively specify the enqueing operation as:

```
enq = proc (q: queue, e: t)
        %@ rest(q) :— rest(q), e()
```

This specification is wrong because, when the queue is empty, the new element becomes *front* and *rest*. The correct specification is thus disappointingly crude, since it cannot distinguish the two ends of the queue. If binding assertions could be made conditional on *emp(q)*, the cases could be separated and the distinction made whenever the queue is known to be non-empty.

Chapter 9

Conclusion

This chapter reports on some experience using Aspect. It relates Aspect to some similar research, and discusses them in the light of the principles that guided the design of Aspect. It ends with a brief assessment of Aspect's contribution and the outlook for the future.

9.1 Experience

Aspect has undergone three iterations. This thesis describes the third version. The first (described in [Jac91]) had no clear model of references between objects, so, amongst other problems, aliasing was not expressible and spurious bugs were reported when a dependency due to a side-effect was not detected. There were abstraction functions, but they were not fully exploited: they were used to compare two specifications of each abstract operation—one in representation terms and one in abstract terms—both of which had to be provided.

I discovered flaws of the first version in a series of small experiments (see Table 9.1). I started by specifying CLU's built-in operations and types. I taught Keith Randall, an undergraduate, how to use Aspect, and together we wrote specifications of some code that had been running without manifesting bugs for several months. This old code was partly from the LP theorem prover [GG89], and partly from the Aspect checker itself. The Aspect specifications of LP were based on detailed informal specifications that had already been written. To our surprise, we found a bug in LP: an unstated precondition was necessary to ensure the correct working of a procedure. We did not discover whether there was a calling context that would have caused the bug

First version, January 1991

Example	Number of lines		Bugs detected	
	Code	Spec	Aspect	Run-time
Built-ins	–	500	–	–
Old code	6500	1700	1	–
New code	1475	720	3	11

Third version, January 1992

Example	Number of lines		Bugs detected	
	Code	Spec	Aspect	Run-time
New LP help	261	96	2	1

Table 9.1: Some figures from small experiments

to manifest itself.

Keith also developed some small programs, specified them with Aspect and ran the checker every time he compiled them. This new code comprised a program to play "Mastermind" (a simple matching game) and a toy relational database. Out of 14 bugs in this code that remained after compilation, 3 were discovered by Aspect. Four more would have been discovered if the checker implementation had been more complete (by handling dependencies on constants). Aspect would then have halved the number of bugs that escaped the type checker.

I had hoped that aliasing would be rare enough that spurious messages would not be a problem. Unfortunately, there were so many that one of the sound messages went unnoticed, lost in the thicket. Unlike Pascal, CLU does not treat aliasing as a rare complication: it is a basic idiom. The standard way to update a data structure is to assign one of its components to a variable and call a procedure with that variable as an argument. The specifications were also far too long (almost half the length of the code).

These two problems spurred the design of the second version of Aspect. The new model introduced pointer aspects, eliminating spurious messages completely. It used abstraction functions to check code against abstract specifications. Combined with new binding assertions, this reduced the size of specifications by about 60%.

Playing with the second version revealed some inconsistencies in the model. The elements of an array were treated as a single object, which led to mistakes in the dependency construction. So I introduced the collection as a second kind of reference aspect.

So far, I have only been able to try the third version on a small example. I built a new help function for LP, which works like the emacs "info" facility. It is implemented in 261 lines of code[1] and uses about 2000 lines of existing LP code. I wrote specifications of my new code (96 lines) as well as the pre-existing abstract types that it uses. I found 2 bugs with Aspect and 1 at runtime.

No strong conclusions can be drawn from this limited experience. It does show, however, that Aspect specifications can be written for real code and that it does find bugs. I think that it would be rash to extrapolate from the first round of experiments that Aspect could detect half the bugs missed by a type checker. But on the other hand, I would expect there to be more Aspect bugs in a larger system developed by a team where interface problems are more common. I suspect also that the writing of Aspect specifications—like type declarations—itself reduces the incidence of bugs, although this theory is hard to test.

9.2 Comparison to Other Schemes

The introduction compared Aspect to its distant cousins such as program verification and run-time assertions. Here we shall look at some closer relatives: techniques using partial specifications that can be analyzed tractably to detect bugs in code.

9.2.1 Inscape

Perry's Inscape system [Per89] is the closest to Aspect in motivation. Its central idea is "constructive specification". Rather than viewing specifications as distinct documents that are justified in their own right, or are used exclusively for program verification, Perry suggests that they be regarded as part of the program structure. This means that they can be exploited by a

[1]Excluding some parsing code that was not specified.

development environment for a variety of purposes (such as change management and version control) that previously relied on information gleaned from the syntactic structure of the program alone.

Inscape's specification language, Instress, is based on Hoare triples. A procedure has a pre-condition that is required to hold on entry and a post-condition that is guaranteed to hold on exit. The termination states can be partitioned into normal and exceptional conditions, with their own pre- and post-states. Instress also provides "post-obligations", which are the post-state analogy to pre-conditions: they specify that some predicate is required to hold later in the execution (for example, that a file eventually be closed).

The pre- and post-conditions and post-obligations are given as formulae in terms of predicates that, like aspects, have no given interpretation and are matched syntactically. For example, a procedure *readFile* might have a pre-condition *FileIsOpen(f)*; this would be discharged if it is preceded in the code by a call to *openFile*, whose post-condition would contain the same predicate. Abstract predicates can be defined in terms of these basic predicates, in a manner akin to Aspect's abstraction function.

The specification of a module can be synthesized from the specifications of its components, by propagating the conditions and obligations. For example, if *readFile* appears in a procedure *p* and is not preceded by another procedure whose post-condition establishes its pre-condition, the predicate *FileIsOpen(f)* will be added to *p*'s pre-conditions. This propagation is performed according to a special logic in which predicates may be known, unknown or possible.

Bug detection is a byproduct of this process. In some cases, propagation fails because a predicate hits a logical barrier, when the predicate of a pre-condition or obligation is contradicted. For example, if *readFile* is preceded by *closeFile*, whose post-condition is $\neg$ *FileIsOpen(f)*, then the pre-condition *FileIsOpen(f)* hits a "ceiling" and can be moved back no further. This suggests a bug in the code. Perry did a study of reported bugs in the development of a large telephone switch and found that a high proportion could be attributed to interface faults of this nature [Per87].

9.2.2 Type Systems

Type systems are a popular research area. Since Algol-60 (the first widely known language to be statically typed), there has been a trend towards type systems that allow more bugs to be detected at compile-time. Abstract typing is move in this direction too. In Algol or Pascal, the type checker will prevent the addition of a character to an integer, but given the declarations

temperature = *int*
weight = *int*

will not prevent the addition of a temperature and a weight. Type equality is determined by structure in these languages, and since temperatures and weights are structurally equivalent, they are not distinguished. In CLU, on the other hand, the two types could be made abstract (by implementing them as clusters). CLU matches abstract types by their names, so even if they had the same representation, they would be deemed different.

LISP and its successors (e.g., Scheme) exemplify a different trend. These languages require no type declarations and perform no static type checking (although, unlike C, primitive types are checked at run-time). They allow programs to be written that are not expressible in CLU or Pascal, such as a function that reverses a list of objects of diverse types.

Type Inference

Recent research attempts to get the best of both worlds: the freedom of LISP and the static checking of CLU. In ML, for example, there are no type declarations, but, by a process known as *type inference*, the compiler constructs the types of larger expressions from smaller expressions. This is not an endless recursion: primitive functions like + have pre-defined types. In general, an expression may have many types. Consider a function that takes another function as its first argument and applies it twice to its second argument:

twice (f, x) = f (f (x))

This function can be applied to argument of different types, for example

twice (square, 2) = 4
twice (not, true) = true.

ML's type inference would assign this function the *polymorphic type* scheme

$$((t \rightarrow t), t) \rightarrow t$$

that denotes a set of types in which the type variable *t* is bound according to the context: *int* in the first example and *bool* in the second. This type system is less restrictive than CLU's; *twice* can be coded in CLU but it cannot be passed as an argument to another procedure before it has been instantiated. But it is also more restrictive than the run-time typing of LISP. A heterogeneous list has no type in ML, for example, and a polymorphic function passed as an argument cannot be instantiated in two ways.

Type inference relieves the programmer of writing declarations, but is a retrograde step for bug-detection. The lack of explicit declarations makes it harder to locate a bug, since the point at which type inference fails may be nowhere near the source of the problem. It also undermines modularity. The inferred type of a function may be consistent but wrong; only when the function is used will the bug be found[2].

Refinement Types

Research in polymorphic types may be seen as an attempt to make the type system less restrictive, accepting more programs as well-typed. In this sense it is antithetical to bug detection, where the value of type checking is in accepting fewer programs. Refinement types [FP92] are a move in the other direction. By elaborating the ML type system with subtypes, more errors can be caught and certain complier optimizations become possible. For example, a *list* type could be refined into two subtypes *empty* and *non-empty*. The functions *first*, which returns the first element of the list, and *new*, which creates a new list, can then be typed as:

first: non-empty $\rightarrow$ *t*
new: $\rightarrow$ *empty*

The expression *first (new ())* now has no type, because one cannot take the first element of an empty list. In general, a function has several types:

append: (empty, t $\rightarrow$ *non-empty)*
$\vee$ *(non-empty, t* $\rightarrow$ *non-empty)*

Refinement types (like regular types in ML) are inferred automatically. Because of higher-order functions, the cost may vary exponentially with the

[2]To alleviate the problems this would cause for team projects, ML does provide a module mechanism that enforces declaration of types.

number of refinements, so a small set must be selected. The refinements are defined by the programmer for the basic constructors of the type and inferred elsewhere. Some types (such as integers) do not have basic constructors, so they cannot be refined.

The programmer may write a refinement type declaration, but if it cannot be inferred it will be rejected. Suppose we have a function that removes all the elements of a list but one. The type inferencer could not prove such a property and so it would not pass a declaration saying that a non-empty list is returned.

Type systems are better thought of as partial verifiers than bug detectors. They cannot admit hints from the user that might undermine their guarantee of no run-time type errors. Bug detectors, on the other hand, can extend their power by accepting, with a leap of faith, assertions made by the user. The Aspect checker infers from a procedure specification that dependencies that are not listed can be assumed to be missing, even if they appear to be present in the code. It can thus incorporate facts that are true but could not have been inferred tractably.

The need to prove all type assertions leads to a more significant problem: an abstract type cannot have refinements that are independent of its representation. One could not define a set type with refinements *empty* and *non-empty* and then implement the set as a hash-table, for example. Refinement types have also only been applied so far to functional programs, although work on imperative extensions is underway.

Typestate

A similar use of typing [Str83] was invented for the NIL programming language. Its purpose was to enforce at compile-time the safety of some low-level language constructs that were notorious causes of bugs. The subtypes in this case were called *typestates.* A variable's typestate may change when acted upon by an operation. The change is determined by the typestate declaration of the operation, which is like a pre- and post-condition giving the typestate the variable should be in when the operation is called and the typestate it will be in when the operation returns. A message type, for example, might have typestates *uninitialized*, *empty* and *full*; the allocate operation would require a message variable to be *uninitialized* and change it to *empty*. Each type of the language has a regular grammar that defines the legal typestate transitions. If a procedure has forbidden transitions, it is rejected by the compiler.

By providing a set of abstract types (such as messages), the designers of NIL hid primitive pointer operations from the programmer and could prevent errors like dereferencing an uninitialized pointer by typestate checking. The typestate system did not allow a variable to have several possible typestates at some point in the program (unlike refinement types discussed above) and when branching in a program led to this, the checker simply deemed it illegal. So one could not write a NIL program in which one branch of an if-statement initialized a message and the other branch did not. This discipline makes sense for low-level types, but for user-defined types—which cannot be given typestates anyway—it would be too constraining.

FX: Side-Effects as Types

The final type system we shall look at is FX [GJSO92]. Like refinement types, its purpose is to make finer distinctions in highly polymorphic code. Its motivation, though, is not bug-detection but exposing parallelism. By regarding the presence or absence of side-effects as part of the type system, FX can determine whether two expressions can be evaluated at the same time. There are three classes of side-effects: read, write and allocate [GL86]. In the same way that compound types can be built out of primitive types, compound effects are built of these and a function constructor. The effects of expressions are given first-class status, and effects, like types, may be polymorphic. Effects can be parameterized by "regions", partitions of the heap [GL88]. Expressions that have side-effects on disjoint objects can then be distinguished (and executed in parallel). Recent work has concentrated on "effect reconstruction": inferring effects without declarations [JG91].

Effect reconstruction can never fail, since every expression has some effect. Effect systems do not immediately help detect bugs, but they may be useful as part of another static analysis. The alias analysis that FX performs implicitly is quite similar to Aspect's. It is less accurate, because regions are essentially sets of objects and thus imply a transitive aliasing relation. If x and y may be aliased, and x and z may be aliased, they will all be placed in the same region, even though y and z may never be aliased. Also, the sequencing of operations can only accumulate effects, so an alias cannot be undone as in Aspect. But effect analysis is simpler and more efficient than the Aspect analysis and extends elegantly to higher-order functions. Perhaps a version of Aspect could be designed that relied on the pre-calculation of effects in the same way that it relies on pre-calculation of types.

9.2.3 Event Sequence Analyzers

Many compilers perform certain "sanity" checks: that a variable is defined before it is read, or that a pointer is initialized before it is dereferenced. One of the earliest and most comprehensive systems to do this kind of analysis was DAVE [FO76]. Three kinds of event are identified: read, define and undefine. Suspicious sequences of events are defined with path expressions; if the events of a variable match one, a message is displayed. For example, the code fragment

$$x := y;\ x := z$$

matches the path expression *dd* representing two definitions without an intervening read and is probably an error since the first has no effect. Because the analysis is performed statically, a match may occur for an "infeasible path"—one that cannot occur in practice—and give a spurious message. It also crosses procedure boundaries, so clever algorithms are needed to make it practical.

It is not clear whether sequencing anomalies arise with any frequency in programs written in modern languages. In CLU, for example, there are no dangling references because the garbage collector deallocates storage automatically; no array initialization errors because the design of the array operations prevents the creation of an array with holes in it; and few variable reference anomalies because of strict scoping rules that encourage declaration of variables at the point of use. The undefine event is a peculiarity of Fortran that cannot arise in CLU: a loop variable may still be in scope after the termination of the loop, but has no defined value. For programs written in low-level languages, these bugs still occur and a tool based on DAVE has been built for C programs[WO85].

Cesar

DAVE detects anomalies rather than bugs: the path expressions are predefined criteria. The idea of catching errors in event sequences can be extend beyond anomalies, by allowing the programmer to define the path expressions. The Cesar system [OO89] lets the user define sequencing constraints, so one can specify, for example, that a file is opened, read and written, and then closed.

Unlike Typestate and Inscape, there are no procedure-level assertions and so the constraints are checked globally across the whole program. This lack of modularity has many unpleasant consequences. Procedures cannot be checked independently without reference to the code of other modules, so none can be checked until all have been coded. A bug in one may mask a bug in another, and bugs cannot be localized within procedure boundaries. Global analysis is also expensive. Apparently, it takes about 3 minutes to check a 20-line program. This would not be a problem in itself, but it appears that the cost varies with the square of the program size, so checking a 1000-line program would take several days. The latest paper [OO92] reports that the implemention is still not fast enough to be applied to real code.

Even if Cesar were to work efficiently, it is not clear that sequencing errors of this nature are very common. Abstract types are usually designed so that their operations can be called in any context without misbehaving.

Comments Analysis and Flavor Assertions

Howden's "comments analysis" [How90] was inspired by the work on Cesar. The programmer writes assertions about the states of objects as formal comments in the code; these are believed by the checker. Assumptions, written at other points in the code, are then matched against assertions for consistency. Flavor analysis [How89] is an instance of comments analysis. A flavor is like a typestate and the assertions take the form "object O has flavor F". An integer used in the summing of an array, for example, might have the flavors *zero*, *partial-sum* and *total.* One could then assert that during the summing the integer is a *partial-sum* and at the end it is a *total.* A point later in the code that relies on the total having been computed could assume that it is a *total.* More elaborate forms of comments analysis allow assertions to make statements about the sequences of operations that have been called up until that point.

Howden's comments are a bit like Inscape assertions, but since they are not associated with procedure boundaries, they call for global analysis. The names of the objects are purely syntactic and the semantics of the programming language plays no role in the comments analysis. Howden sees this as an advantage, because it allows the programmer to make assertions that are not associated with any data object. But it also means that an assertion about an array element *a[i]* has no more relevance to another element *a[j]* than to any other variable; the user must provide all these associations, making separate claims for different syntactic expressions that are semantically identical.

I think it is a mistake to spurn the semantics of the programming language. It results in a loss of compositionality. The specification of a composite structure should be a simple function of the specifications of its parts. Type systems exemplify this quality, and Aspect's compositionality comes by piggy-backing its specifications on top of types. For example, if you declare a type to be an array of records, no new specification is needed, because Aspect puts together the array and record specifications in the right way. Without any semantic common ground, the constructors of the programming language will have no counterparts in the specification domain.

9.2.4 Program Slicing

Program slicing [Wei84] is not a bug detection scheme but its use of dependency analysis is closely related to Aspect's. A program is sliced by choosing an instance of a variable (an appearance on some line of the program) and then deleting some statements that do not affect its value. The resulting skeleton, called a *slice*, is itself a program: an abstraction of the original program whose behaviour is identical on the value of that variable instance. For a programmer wanting to understand why a variable has the wrong value at some point in the program, a slice provides as much information as the whole program but is usually much smaller.

Program slices can be calculated by tracing the dependencies of the variable instance back through the code, marking the lines that might contribute to its value. For the same reason that the Aspect dependency calculation is only approximate, finding the smallest slice is in genera' not possible.

The Aspect calculation could easily be adapted to calculating program slices. The tags of an aspect would become pairs consisting of a source (an aspect of the pre-state upon which that aspect depends) and a line number (the line at which the dependency was established). Using Aspect would give smaller slices than a conventional calculation. For example, in the code fragment

```
e := f
array[t]$addh (a, e)
a[i] := j
s := array[t]$size (a)
```

a slice on *s* in the last line would eliminate all other lines except the second, while a conventional slicing could eliminate none. Programs could also be sliced on aspects rather than variables. A slice on *low(a)* in the last line, for example, would eliminate the first three; a slice on *el(a)* would eliminate none.

9.3 Aspect's Design Principles

Aspect's design was guided from the start by a number of design principles. Here are some of the qualities that I thought were important for the checker and the notation:

- **Soundness**. Aspect never reports a bug when the code could meet its specification. None of the other systems discussed above have this property: they all give spurious messages. The flip side of this coin is that Aspect, in contrast to the tools based on type checking, does not certify any properties of the program.

- **Generality**. Aspect was designed to be applicable to a broad class of programming languages. This meant handling all the typical features of imperative programming that complicate static analysis but are widely used, such as pointers, aliasing, exceptions and dynamic allocation. Aliasing is the most troublesome of these, and is ignored by most other schemes. It is a cruel irony of bug detection that complications in the programming language are especially important, because they are often the cause of bugs.

- **Declarativeness & Referential Transparency**. The order of assertions in a specification should not matter, and the meaning of an expression should be the same wherever it appears. This is true of all the schemes discussed above but was almost not true of Aspect. Prenaming saved the day, by preventing binding assertions from changing the meaning of object names.

- **Predictable Efficiency.** Aspect is roughly linear in the length of the program and in the worst case (nested loops) the time varies with the square of the number of aspects. Its performance does not seem to vary much over the range of programs I have tried it on.

9.4 Contributions

Aspect is a simple scheme: its notation is spare and its mechanism is a simple adaptation of dependency analysis. Yet it achieves some power by bringing together four novel ideas:

1. **Missing Dependencies.** The bugs that result in missing dependencies are mostly not detectable by other static means. Errors of omission, which have not been easy to detect up until now, are particularly amenable to Aspect's analysis, and complement the class of bugs that are readily detectable by type checking. Specifying minimal rather than maximal dependencies has the great advantage of soundness: no spurious bugs are reported.

   Dependency specifications take tractable analysis a step towards conventional specifications. The close relationship between saying "x is determined by y" and "x is a function f of y" makes them fairly natural and easy to write. Because dependency specifications relate different objects, Aspect can check not only what happened to a single object, but how it was put together from other objects. This idea is not new in formal specifications, but it appears to be novel for a tractable scheme.

2. **Aspects.** Dividing objects into aspects brings the dependency calculation to life. Programs built from abstract types tend to have few variables or complex structures in a given scope; most of the interesting structure is hidden, and thus escapes a simple dependency analysis. Aspects introduce structure without exposing the representations of types, leading the simple dependency calculation into performing an abstract analysis about the properties of interest.

3. **Reference Aspects.** Interpreting some aspects as references allows the object structure of the program state (and thus aliasing) to be described

in the same framework. An aspect's possible values and its dependencies fit smoothly together and their calculations are approximated in the same way. The framework subsumes unbounded collection objects too, by viewing their contents in terms of an archetypal element.

4. **Abstraction Functions.** The user's freedom to invent the aspects of a type is the key to the abstract analysis. At the same time, however, freeing the user from the details of the representation is potentially dangerous, since the abstract assertions might not tally with the details they claim to hide. Abstraction functions bridge this gap. Aspect seems to be unique among static analyses in supporting consistency checks across levels of abstraction.

I started this work hoping to substantiate three theses:

- *partial specification*—that a formal specification may be more effective if its scope is limited;
- *abstraction*—that a tractable analysis need not be low-level, and that abstract claims can be checked against concrete details; and
- *credulous checking*—that a checker's power is amplified if it can incorporate information that could not be derived automatically.

My experience with Aspect suggests that together these form a promising approach. The world of routine static analysis performed by compilers and the world of formal specification have existed mostly in isolation from one another. I believe that their fusion holds great opportunities and that this approach is a start in that direction.

Glossary

abstract type The type of a data object is said to be abstract if there is only a fixed set of procedures that have access to its representation. These procedures are usually encapsulated within a module, so that any change to the representation of the type that nevertheless preserves the behaviour of the access operations affects only the code within the module. The CLU module for abstract types is called the *cluster*, and the type checker can (most of the time) prevent outside access to the representation.

abstraction A process of removing accidental information and leaving the essence. The term also sometimes describes the result of the process. A local MIT dialect (but not this thesis) treats the word as a synonym for program module (e.g., procedural abstraction, type abstraction). The keyword 'abstraction' in an Aspect specification marks the abstraction function, which relates the concrete and abstract aspects of a type.

affected by The dependency assertion $\alpha \leftarrow \beta$ is read 'α is affected by β', and means that, in addition to α depending on β, α is read and β is written.

aliasing Two variables are aliased when they are bound to the same location, so that a change to one appears also to be a change to the other. This happens in Pascal, for example, when two call-by-reference actuals are the same variable. It also arises because of pointers, and in CLU, since a variable is a pointer to an object, even plain assignment causes aliasing. I use the term more generally too, whenever a single object or location may be accessed under different names.

allocation Placing a newly created object in an unused area of the program heap. Aspect provides allocation assertions to describe how procedures allocate new objects. The assertion $a : t$ means that a fresh object of type t is allocated and given the name a, local to the procedure specification.

anomaly A flaw in a program that is evident without any idea of what the program is supposed to do, such as reading an uninitialized variable or looping forever. Anomalies are not necessarily bugs (as Lint users know to their cost). Dead code, for example, may do no damage.

argument A variable that names an object on which a procedure acts; distinct from a parameter, which is bound statically in CLU.

array CLU arrays are dynamic and can grow and shrink at either end. The array operations are designed so that an array cannot be created unless all its elements are well-defined. An array is not an indexed set of variables but an object in its own right. The statement *a[i] := j* is a shorthand for the procedure call *array[t]$store (a, i, j)*, and *i := a[j]* likewise is short for *i := array[t]$fetch (a, j)*.

Aspect The position of planets and stars with respect to one another, held by astrologers to influence human affairs. Also, a notation and a technique for detecting bugs in programs.

aspect A name for an abstract component of an object. A *window* object, for example, might have the aspects *colour*, *dimensions*, *position* and *contents*. There are two kinds: *plain* and *reference* aspects. Reference aspects are further divided into *pointer* and *collection* aspects.

aspect expression An aspect of a particular object; *colour(w)*, for example, to denote the colour of a window object called *w*. An aspect expression may be formed from any kind of aspect. If p is a pointer aspect, *p(x)* denotes the identity of location of the object pointed to by the p aspect of x. If c is a collection aspect, *c(x)* denotes the set of identities or locations of the objects referred to by the c aspect of x.

aspect state Aspect specifications are defined over abstract states called 'aspect states'; the checker performs an abstract interpretation over a collecting state corresponding to a set of aspect states. Not to be confused with the state of Aspect, which has been dismal at times.

assertion A clause in an Aspect specification of a procedure. There are three kinds: allocations, bindings and dependencies.

assignment The CLU assignment $x := y$ makes the variable x a name for the object called y. As a result, x and y become aliased: they name the same object, and a mutation of that object under one name will be visible under the other too.

binding A kind of Aspect assertion; $a(p)$:– $b(q)$ says that the a aspect of the object called p (in the pre-state) has the value in the post-state that the b aspect of q has in the pre-state, in some invocation of the procedure.

bug An inadvertent flaw in a program that causes it to misbehave under certain conditions. Most programs contain bugs; they may be hard to identify because they only manifest themselves in rare circumstances, or more commonly, because nobody knows what the program is supposed to do anyway.

checker The Aspect checker is a tool that reads in a CLU program annotated with Aspect specifications and generates a report listing bugs that have been detected.

client A client of a program module is another module that uses it and relies on its correctness. If a procedure p calls a procedure q, for example, then p is a client of q.

clipboard Part of an editor buffer that holds text that has been cut and can be reinserted later.

CLU A programming language invented by Liskov at MIT in the late 1970's. It looks like Algol, but its semantic model is closest to LISP or Scheme. CLU does not stand for 'clumsy' or 'clutter'.

cluster The CLU model for abstract data types. A polymorphic type is built by parameterizing a cluster with a type variable.

collection aspect Represents a set of references. For example, an object a of type *array[t]* has a collection aspect *el*; *a.el* is a name for an element object and *el(a)* is an aspect expression denoting the set of identities of the element objects.

control dependency A standard term from compiler analysis. In the statement *if b then x := y*, x has a dataflow dependence on y and a control dependence on b.

cursor The current position in an editor buffer at which a character typed at the keyboard will be inserted. Called the *point* in emacs.

cvt A CLU keyword short for 'convert' used as the name of a type in the header of an abstract operation. From outside the operation, it denotes the abstract type; from inside, the representation type.

dependence, dependency An aspect α depends on an aspect β over some program text if the initial value of β affects the final value of α. Then α has a dependence on β and there is a dependency between α and β.

dereference In programming language jargon, dereferencing or following a pointer means obtaining the object it refers to.

environment Part of the program state; maps variables to the locations of the objects they denote. In CLU, the environment is a stack, but since the effect of a procedure call can always be determined from its specification, Aspect has no need of a stack and a simple mapping suffices.

handler A handler is a synactic construct in CLU associated with one or more exceptions. It is attached to a simple or compound statement. When a procedure raises an exception, control passes to the handler with the smallest scope that includes the procedure call. The statements of the handler are then executed, and control returns to the point immediately following the statement to which the handler is attached.

immutable An object is immutable if its value can never change over its lifetime. In CLU, a type is either mutable or immutable, and the objects of an immutable type are immutable. Integers, for example, are immutable. The increment $x := x + 1$ causes x to refer to a fresh integer object that is one greater than the object it referred to previously; the integer object itself is not mutated. In a program constructed only from immutable types, aliasing is invisible and assignment may be thought of as copying.

implementation The code of a procedure is one of its implementations.

mark A kind of secondary cursor, used in some editors to delineate a portion of text called the 'region'.

mutable An object is mutable if its value can change; otherwise it is immutable. Arrays are mutable, but integers are immutable.

operation An operation of a datatype is one of its procedures.

ordering The aspect ordering of a type defines, for its set of aspects, which may be derived from which. It is given as an annotation following the aspect declaration.

parameter In CLU, procedures and types may have parameters that are instantiated at compile-time. For example, a sort procedure may have a type parameter t denoting the type of the elements of the array to be sorted; a stack type may have a type parameter for the type of the objects it contains.

plain aspect An aspect that is not a reference aspect.

pointer aspect Represents a reference to a single object. For example, if r is an object of type *record [f:t, g:t]*, then r has pointer aspects f and g. The aspect expression $f(r)$ denotes the identity of the object pointed to by the f field of r; $r.f$ is a name for that object.

polymorphism A procedure is polymorphic if the same code can be applied to arguments of different types: a sorting routine, for example, may work for integers or strings. An abstract type is polymorphic if its objects contain objects of some unspecified type, and of its operations can be applied irrespective of that type. For example, you can implement a type *set[t]* in CLU and instantiate the type variable t to be an integer giving *set[int]* or a string giving *set[string]*. CLU's polymorphism is 'parametric'; there is no subtype polymorphism.

pre, post The pre-state of a procedure is the initial state in which it is invoked; the post-state is the final state immediately following its execution.

prenaming Since the naming of objects may change when a procedure is executed, some convention is needed for the naming of objects in assertions. Objects are 'prenamed' in Aspect, which means that any object name appearing in an aspect expression refers to an object named in the pre-state of any procedure call.

procedure May return an object in CLU and need not cause side-effects on its arguments.

record An object that contains a fixed set of named references to other objects. A CLU record is like a struct in C in which the fields are pointers.

reference aspect Represents one or more references to other objects. There are two kinds of reference aspects: pointers (for single references) and collections (for multiple references).

region The portion of text in an editor buffer between the mark and the cursor.

rep A CLU keyword for the representation type of a cluster. A cluster always implements exactly one abstract type, so *rep* is unambiguous.

representation The type in which an abstract data type is implemented. A set abstract type, for example, may be implemented as a binary tree, a hash table or an array: these are all possible representations of a set. A CLU abstract type may only have one representation, which is defined in the cluster of the type.

rep exposure The representation of an abstract type is exposed if one of its operations is coded in such a way that part of the representation becomes accessible outside the cluster. This is bad news, because the operations can no longer be reasoned about in isolation.

signature The header of a procedure gives the names and types of its arguments and results. In type checking calls to the procedure, only the signature is needed: the header without the argument names.

sound A sound bug detection scheme is one never reports a bug unless a bug is present, that is, the code does not meet its specification. We use the term in its logical sense, where the inference is considered to be the reporting of a bug.

zapping Deleting from the text of a buffer all the characters between the buffer and the next occurrence of some character. The deleted text is placed in the clipboard.

Bibliography

[Acr79] A.T. Acree, T.A. Budd, R.A. DeMillo, R.J. Lipton and F.G. Sayward, *Mutation Analysis*, Tech. Rep. GIT-ICS-79/08, School of Information and Computer Science, Georgia Inst. of Technology, Atlanta, GA, Sept. 1979.

[BC85] Jean-Francois Bergeretti and Bernard A. Carre, 'Information-Flow and Data-Flow Analysis of while-Programs', *ACM Trans. Programming Languages and Systems*, 7/1, Jan. 1985.

[CHT79] Thomas E. Cheatham Jr, Glenn H. Holloway and Judy A. Townley, 'Symbolic Evaluation and the Analysis of Programs', *IEEE Trans. on Software Eng.*, SE-5/4, July 1979.

[CWZ90] David R. Chase, Mark Wegman and F. Kenneth Zadeck, 'Analysis of Pointers and Structures', *Proc. ACM Conf. on Principles of Programming Language Design and Implementation*, 1990.

[Dij76] E.W. Dijkstra, *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, 1976.

[Fer87] J. Ferrante, 'The Program Dependence Graph and Its Use in Optimization', *ACM Trans. Programming Languages and Systems*, July 1987.

[FO76] L.D. Fosdick and L.J. Osterweil, 'Data Flow Analysis In Software Reliability', *ACM Computing Surveys*, 8/3, Sept. 1976.

[FP92] Tim Freeman and Frank Pfenning, 'Refinement Types for ML', *Proc. ACM Conf. on Principles of Programming Language Design and Implementation*, 1991.

[GG89] Stephen J. Garland and John V. Guttag, 'An Overview of LP, the Larch Prover', *Proc. 3rd Int. Conf. on Rewriting Techniques and Applications*, Chapel Hill, N.C., Lecture Notes in Computer Science 355, Springer-Verlag, 1989.

[GJSO92] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon and James O'Toole, *Report on the FX-91 Programming Language*, MIT/LCS/TR-531, Feb. 1992, MIT Lab For Computer Science, Cambridge, MA 02139.

[GL86] David K. Gifford and John M. Lucassen, 'Integrating Functional and Imperative Programming', *ACM Conf. on LISP and Functional Programming*, Aug. 1986.

[GL88] David K. Gifford and John M. Lucassen, 'Polymorphic Effect Systems', *Proc. ACM Conf. on Principles of Programming Languages*, 1988.

[Gut77] John V. Guttag, 'Abstract Data Types and The Development of Data Structures', *Comm. ACM*, 20/6, June 1977.

[GY76] Susan L. Gerhart and Lawrence Yelowitz, 'Observations of Fallibility in Applications of Modern Programming Methodologies', *IEEE Trans. on Software Eng.*, SE-21/3, Sept. 1976.

[Hoa72] C.A.R. Hoare, 'Proof of Correctness of Data Representations', *Acta Informatica*, 2, 1973.

[How76] William E. Howden, 'Reliability of the Path Analysis Testing Strategy', *IEEE Trans. on Software Eng.*, SE-2, Sept. 1976.

[How77] William E. Howden, 'Symbolic Testing and the DISSECT Symbolic Evaluation System', *IEEE Trans. on Software Eng.*, SE-3/4, July 1977.

[How89] William E. Howden, 'Validating Programs Without Specifications', *Proc. 3rd ACM Symp. on Software Testing, Analysis and Verification (TAV3)*, Key West, Florida, Dec. 1989.

[How90] William E. Howden, 'Comments Analysis and Programming Errors', *IEEE Trans. on Software Eng.*, SE-16/1, Jan. 1990.

[HPR89] Susan Horwitz, Phil Pfeiffer and Thomas Reps, 'Dependency Analysis for Pointer Variables', *Proc. ACM Conf. on Principles of Programming Language Design and Implementation*, 1989.

[Jac91] Daniel Jackson, 'Aspect: An Economical Bug-Detector', *Proc. 13th Int. Conf. on Software Eng.*, Austin, Texas, May 1991.

[JG91] Pierre Jouvelot and David K. Gifford, 'Algebraic Reconstruction of Types and Effects', *Proc. ACM Conf. on Principles of Programming Languages*, 1991.

[Lam81] B. W. Lampson et al., *Report on the Programming Language Euclid*, Report CSL-81-12, Xerox PARC, CA, October 1981.

[LG86] Barbara Liskov and John Guttag, *Abstraction and Specification in Program Development*, McGraw Hill, 1986.

[LH88] James R. Larus and Paul N. Hilfinger, 'Detecting Conflicts Between Structure Accesses' *Proc. ACM Conf. on Principles of Programming Language Design and Implementation*, June 1988.

[Lis79] Barbara Liskov et al., *CLU Reference Manual*, Oct. 1979, MIT/LCS/TR-225, MIT Laboratory For Computer Science, Cambridge, MA 02139.

[Lis92] Barbara Liskov, 'A History of CLU', to appear in *ACM Symp. on the History of Programming Languages*, 1992.

[LvH85] D.C. Luckham and Friedrich W. von Henke, 'An Overview of Anna, A Specification Language for Ada', *IEEE Software*, 2/2. March 1985.

[Mey88] B. Meyer, *Object-oriented Software Construction*, Prentice-Hall, 1988.

[Off92] A. J. Offutt, 'Investigations of the Software Coupling Effect', *ACM Trans. on Software Engineering and Methodology*, 1/1, Jan. 1992.

[OO89] Kurt M. Olender and Leon J. Osterweil, 'Cesar: A Static Sequencing Constraint Analyzer', *Proc. 3rd ACM Symp. on Software Testing, Analysis and Verification (TAV3)*, Key West, Florida, Dec. 1989.

[OO92] Kurt M. Olender and Leon J. Osterweil, Interprocedural Static Analysis of Sequencing Constraints', *ACM Trans. on Software Engineering and Methodology*, 1/1, Jan. 1992.

[Par72] David L. Parnas, 'On the Criteria to be Used in Decomposing Systems into Modules', *Comm. ACM*, Dec, 1972.

[Per87] D.E. Perry and W.M. Evangelist, 'An Empirical Study of Software Interface Faults—An Update', Proc. of 20th Annual Hawaii Conf. on System Sciences, Jan. 1987.

[Per89] Dewayne E. Perry, 'The Inscape Environment', *Proc. 11th Int. Conf. on Software Eng.*, May 1989.

[Per89] Dewayne E. Perry, 'The Logic of Propagation in The Inscape Environment', *Proc. 3rd ACM Symp. on Software Testing, Analysis and Verification (TAV3)*, Key West, Florida, Dec. 1989.

[Str83] Robert E. Strom, 'Mechanisms for Compile-time Enforcement of Security', *Proc. ACM Conf. on Principles of Programming Languages*, 1983.

[Suf82] Bernard Sufrin, 'Formal specification of a Display-Oriented Text Editor', *Science of Computer Programming*, 1982, Vol. 1, pp. 157–202.

[Wei84] Mark Weiser, 'Program Slicing', *IEEE Trans. on Software Eng.*, SE-10/4, July 1984.

[WO85] Cindy Wilson and Leon J. Osterweil, 'Omega—A Data Flow Analysis Tool for the C Programming Language', *IEEE Trans. on Software Eng.*, SE-11/9, Sept. 1985.

DARPA OFFICIAL DISTRIBUTION LIST

DIRECTOR Information Processing Techniques Office Defense Advanced Research Projects Agency (DARPA) 1400 Wilson Boulevard Arlington, VA 22209	2 copies
OFFICE OF NAVAL RESEARCH 800 North Quincy Street Arlington, VA 22217 Attn: Dr. Gary Koop, Code 433	2 copies
DIRECTOR, CODE 2627 Naval Research Laboratory Washington, DC 20375	6 copies
DEFENSE TECHNICAL INFORMATION CENTER Cameron Station Alexandria, VA 22314	2 copies
NATIONAL SCIENCE FOUNDATION Office of Computing Activities 1800 G. Street, N.W. Washington, DC 20550 Attn: Program Director	2 copies
HEAD, CODE 38 Research Department Naval Weapons Center China Lake, CA 93555	1 copy